ESD-TR-72-121 Vol. 2



# A STUDY OF FUNDAMENTAL FACTORS UNDERLYING SOFTWARE MAINTENANCE PROBLEMS: FINAL REPORT

APPENDICES

20 December 1971



DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730



(Prepared under Contract No. F19628-71-C-0125 by Corporation for Information Systems Research and Development/CIRAD, 401 N. Harvard, Claremont, California 91711.)

ESD-TR-72-121

# A STUDY OF FUNDAMENTAL FACTORS UNDERLYING SOFTWARE MAINTENANCE PROBLEMS: FINAL REPORT

Appendices

20 December 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730



(Prepared under Contract No. F19628-71-C-0125 by Corporation for Information Systems Research and Development/CIRAD, 401 N. Harvard, Claremont, California 91711.)

## APPENDICES

| APPENDIX | TITLE |
|---|---|
| I | Statement of Work for This Study |
| II | Software Maintenance: Questions and Answers |
| III | BUIC Case Studies and Diary |
| IV | A Study of Factors Inhibiting the Effectiveness of Maintenance Programmers at Chrysler Corporation |
| V | CSA Software Maintenance Report |
| VI | - Scenarios and Questions<br>- Higher Order Languages and Maintainability |
| VII | Technical Approach and Aims for a Path Analysis Feasibility Study |
| VIII | Instructions for Path Analysis Experimental Programmer |
| IX | Specification of Modifications for Path Analysis |
| X | Minimum Paths According to Staff Programmer |
| XI | Experimental Programmer's Verbalization While Making Modifications |
| XII | Guidelines for Keeping a Magnetic Tape Log of Program Maintenance Mental Processes |
| XIII | Tape Logs of Experimental Programmers |
| XIV | Language Statement Types Which Define Conceptual Groups |

# APPENDIX I

## Statement of Work

## for This Study

# STATEMENT OF WORK

## Computer Aided Software Maintenance Study

1.0 Introduction: Systems programmers must often maintain programs that someone else wrote. To do so, they must first learn about these programs by consulting the written documentation prepared by the original programmer. Such documentation commonly consists of flowcharts at various levels of detail, narrative descriptions, and ultimately, program symbolics. Even for only moderately complex systems, it is difficult to use these materials efficiently to remedy defects that show up in use or to evaluate proposed system changes.

In this procurement, the Contractor will focus on problems faced by programmers who must maintain programs someone else wrote. He will identify and study the factors which inhibit the effectiveness of current maintenance programming aids, and as a result of this study, he will propose new kinds of computer aids for use by maintenance programmers. The emphasis in this study will be on the development of principles underlying the effective use of such aids, although some effort will be devoted to initial development and test of promising aids.

2.0 Scope

2.1 Objective: The objective of this procurement is to study problems of maintaining complex programs in order to develop more effective, computer aids for software maintenance. It is intended that these methods be especially useful to programmers who must maintain programs that someone else wrote.

2.2 Approach: Contractor will investigate fundamental problems limiting the effectiveness of maintenance programmers and will propose and study new techniques for increasing their effectiveness. He will identify inadequacies in current methods and identify reasons for these inadequacies. He will develop case studies illustrating principles and problems encountered in software maintenance, together with some estimate of the importance of the principles and problems. He will specify and carry out a research program for developing further information that must be known in order to create useful maintenance programming aids, and he will specify and investigate new methods to help maintenance programmers. He will investigate the role of graphics consoles in maintenance programming. At the end of the study, he will present a balanced view of realities vs. possible techniques, together with a plan for further study of these problems and techniques.

3.0 Areas of Consideration

3.1 Use of Graphics Terminals: Contractor's study should emphasize the use of on-line interaction with a graphics console.

3.2 Case Studies: The Contractor will develop and evaluate his ideas with respect to the maintenance or modification of particular compilers, operating systems, data management systems, or other sets of existing complex programs with which he is familiar. From this source of material, the Contractor will draw examples of the kinds of problems one faces in maintaining or modifying programs that someone else wrote. The Contractor will evaluate his proposed techniques with respect to these specific real-world examples. Preferably the system of programs that serves as a test environment and as a source of case studies will be a system that is already being maintained or modified by the Contractor. Note that it is not necessary and not expected that any software modifications or software maintenance be carried out. The Government only desires that the research be accomplished with respect to actual problems that have arisen in connection with existing systems of programs.

3.3 Extent of Programming Effort: Contractor shall test his techniques by thoughtful, scientific study; he shall not embark on an extensive programming effort before fundamental limitations have been carefully identified. The Contractor is encouraged to test his ideas on actual computers only if a relatively small amount of programming is required for a meaningful test.

3.4 Possible Problem Area (example): One of the inherent deficiencies in current documentation methods is the presentation of information only in a static format. A programmer who must analyze program flow for complex test cases may have considerable difficulty in relating the test case to the actions described in the flowcharts. Hence means might be provided to show the actual program flow (on-line) for test cases specified by the programmer, and thus the programmer might learn the structure and function of the program more quickly and more thoroughly.

3.5 Interdisciplinary Approach: The Contractor is encouraged to include on his research team a person with background in the behavioral sciences as well as persons experienced in computer sciences. Such a person will be expected to help devise suitable methods for scientifically assessing the general advantages and limitations of the techniques to be developed under this Contract. He would be expected to provide useful insight into the cognitive processes and the inherent human limitations and requirements of maintenance programmers.

4.0 Task to be Accomplished

4.1 Phase I - Overall Analysis:

4.1.1 Step 1: The contractor will identify problems interhent in (and

especially those peculiar to) maintenance programming where the maintenance programmer is not the original programmer. The maintenance programmer is assumed to be an experienced systems programmer, and the system being maintained is to be considered fairly complex (e.g., any system which would normally be programmed by more than one programmer).

4.1.2 Step 2: A set of case studies, based on actual systems maintenance problems (see 3.2 above) will be developed to illustrate the fundamental principles and the problems inherent in maintenance programming. (The case studies are to serve as paradigms of system maintenance problems, and hence should represent problems which are in some sense typical of the problems faced by maintenance programmers.)

4.1.3 Step 3: A number of possible maintenance programming computer aids will be proposed. Each proposed aid will be characterized in terms of (1) its relation to fundamental factors underlying systems maintenance problems; (2) the kind of research that must be accomplished to assess the probable utility (e.g., what information must be known to make this assessment, and how could this information be developed) of the proposed aid; and (3) the amount of work that would be required to test the proposed aid's effectiveness. (The solutions proposed in this part of Phase I, together with the case studies, will be used by the Air Force to determine the extent and value of initiating a broader, more intensive, research effort on maintenance problems). Solutions proposed as part of this step will range from conservative to highly speculative, since the purpose of the Step is, in part, to stimulate thought on solutions to maintenance programming problems.

4.1.4 Step 4: The contractor will next select a small set of problems, factors, and/or proposed solutions to be investigated more thoroughly in the remainder of the contract (Phase II). The rationale for this selection and a plan of research for gaining a deeper understanding of the selected issues will be furnished (see 4.2 and 4.3 below).

4.2 Phase I - Report: The results of the Phase I study will be presented in a Technical Report consisting of two parts. Part I will contain the results of Steps 1, 2, and 3 of Phase I. Part II will contain the information developed in Phase I, Step 4.

4.3 Phase II - Intensive Study of Selected Problems/Solutions: A scientific study of selected problems or proposed solutions to software maintenance will be undertaken as specified in Part II of the Phase I report (see 4.2). The purpose of this study is to examine more carefully the feasibility of these approaches and especially, the underlying fundamental problems that might impede the effectiveness of proposed techniques. The emphasis of the study will be the development and evaluation of principles concerning the nature of software maintenance so that the effectiveness of possible programmer aids can be more accurately assessed in advance of implementation or actual tests. Limited evaluation of proposed programmer aids will be accomplished (see 3.2). Extensive programming will not be undertaken

in this phase. Instead, scientific study and pilot experiments together with judicious hand-simulation or analysis of proposed computer implementations will be used insofar as possible to assess the probable effectiveness of proposed techniques and to examine underlying problems. (Some programming effort will undoubtedly be necessary, but since the emphasis of this contract is the study of principles underlying the effectiveness of proposed maintenance programmer aids rather than the implementation of immediately useful aids, it is expected that a relatively small amount of programming effort will be required.)

4.4 Final Report: At the conclusion of Phase II (4.3), a Final Report will be prepared consisting of 1) Part I of the Phase I report, revised as necessary to reflect knowledge gained in the Phase II studies; and 2) a presentation of results obtained in Phase II.

4.5 Part II of the Phase I report (4.2) will serve as a working paper defining the work to be accomplished during the remainder of the contract (Phase II). Work on Phase II will not proceed until receipt of Government approval. A decision on whether or not the proposed plan is approved, with or without modifications, will be rendered within seven (7) calendar days of the receipt of Part II.

# APPENDIX II

## Software Maintenance: Questions and Answers

CONTENTS

# PART I

## INTRODUCTION AND INTERVIEWEES

### INTRODUCTION

This document is a compendium of answers to various questions about the problems of maintaining software systems. These answers were generated during interviews over the period of 2 March to 6 April 1971. The interviews generally were conducted by a two-man team with one person reading questions from a prepared questionnaire and the other injecting clarifying inquiries while taking notes. The results viewed as a table of interviewee versus questions are not complete because some questions did not apply to specific interviewees. In other cases, certain questions evoked such a torrent of freely flowing comment that it seemed best to pursue these lines to the neglect of the prepared questions. The emphasis was on obtaining realistic information and opinion directly and indirectly related to the problem of maintenance programming.

### THE INTERVIEWEES

Five full interviews were conducted. The people interviewed are listed below with some background material about each. A sixth person, Michael Castin, commented on some of the questions.

ROBERT HARRINGTON - Mr. Harrington was in charge of a staff of 260 including 100 programmers, who used 21 computers, at Chrysler Corporation's centralized computing facility in Detroit. A large part of the effort involved here concerned a real-time on-line order entry system for control of assembly lines. Mr. Harrington was administered a different questionnaire than the other four. This questionnaire differed in the order and emphasis of the questions asked. His responses have been rearranged to fit the format of the other questionnaire using the best judgment of the authors.

JOHN BROWN - At TRW Mr. Brown's group is "the organizational focal point for questions regarding software maintenance."

PAUL SLEEPER - Mr. Sleeper is Director of Technical Development for Remote Computing Corporation. His group of seven people is "responsible for the technical products that the company offers." His primary duties are the development and maintenance of

executive programs for a time-sharing system. Secondly, he contributes to the development of some business application programs. His company's computing center is organized around two Burroughs 5500s. Each can theoretically handle around 40 terminals. A total of around 50 terminals, including several used in-house by the company, are actually serviced by the two 5500s.

WILLIS HUDSON - Mr. Hudson is presently an employee of CSA. He is experienced in machine-language programming and in data reduction. He also was employed at SDC in the development and maintenance of SAGE programs. Later assignments at SDC included developing and maintaining software which "solved the hardware interface"; e.g., compilers, and executive monitors. He was also employed for a time at Jacobi Systems. Mr. Hudson will graduate from law school in June and is preparing for the Bar Exam.

DANIEL COVILL - Mr. Covill is Associate Director--Development, Computer Center at the University of California at San Diego. Previously he was the Chrysler Parts Division's Manager of Programming. There a major area of concern was the maintenance of programs which others had written. His employment prior to Chrysler was with the University of Wisconsin, the Burroughs Corp., and SDC on the SAGE project.

MICHAEL CASTIN - Mr. Castin, an employee of SDC, maintains and directs the maintenance of portions of the BUIC software system. He has also used his extensive BUIC experience in consulting on the present project.

## INTRODUCTORY RESPONSES TO QUESTIONS

Question: First, we would like some general background information on the overall system you are working on and what you do. What is the overall purpose of this software system?
Multimachine ?
Programmed in an HOL?

Mr. Covill: On the SAGE project, the system was well defined. But at Chrysler, it was "not clear what the system was . . . a lot of systems for one customer." A programmer felt he was working on a program, not on part of a system: there was just a large collection of batch-mode programs.

"The government contractor thing is better, for maintenance programmers, than the business thing."

Mr. Hudson: At Jacobi Systems, the purpose was the effective use of a Univac 1108. The operating system was Exec-2. Interfaces included those to applications-oriented programs; the concerned IO, and a library of subroutines.

It was not a multi-machine system, it was batch only, and it was not programmed in a higher-order language.

Mr. Sleeper: It is: time-sharing, primarily;

multi-machine;

programmed as much as possible in higher-level languages.

Mr. Brown: In addition to serving in a consulting capacity in TRW, Mr. Brown's group is working on "PACE (Product Assurance, Checkout and Evaluation)." This is, or will be, a system written in Fortran for assisting in the check-out of other programs written in Fortran.

Only one program, composed of about 1,000 statements, is actually operational in PACE. Five other programs are planned.

The one operational program "will look at a program, and find all ways--all directed graphs--through it . . . it will label chords." The result is "a nice representation of the logical structure of the program under the statements."

(Our Comment: This part of PACE seems to have some close antecedents in the work of others, namely Green, 1970.)

Question: Does it do this before compilation?

Mr. Brown: Yes. It "looks at the program, modifies it" (by adding a statistical table) and "evaluates the percent of the code that has been exercised by the current test data."

Question: How large a project is PACE?

Mr. Brown: It is a sort of "pet project." But, in terms of memory, "the TS core limit is 64K octal." In terms of people, four people have worked on PACE off and on for about a year, for a total of only about six man-months of effort.

## PART II

## SOURCES OF REFERENCE INFORMATION

Question: Where do you get the information you need to make modifications or correct the errors? More specifically, please comment on the value to you of the information to be gotten from . . .
The request itself . . .

Mr. Castin: "There is no substitute for a well-explained request."

Mr. Hudson: There were some sophisticated users who made "very helpful" requests, but most were "simply FORTRAN programmers; they would say, 'This program ran before, and I just changed two cards.'" These were not very helpful.

Mr. Sleeper: In general there are two very different kinds of requests. One complains of a Command and Edit failure. The information given "is usually pretty good," although, of course, it varies with the source. "I insist on the teletype sheets (which show the effect) of the 'disaster,' and listings" from all parties.

A second kind of request is a vague request for some kind of extension of the system. These are not usually very useful.

Mr. Brown: It's very valuable for developing a program, and in "telling you where to go to change the coding." For example, PACE is being adapted to a 360; the requests come in the form of specifics such as "You have to use BLOCKDATA subroutines."

(Our Comment: Since PACE is not an operational system, these are not really maintenance requests; they are pieces of advice, from one programmer to another, during the development of a program.)

Question: . . . the value . . . of the information . . . from written documentation?

Mr. Hudson: For Exec 2, "it varied from extremely bad to non-existent."

Mr. Sleeper: "That is an interesting question. With the 5500, it is mostly oral; there is not all that much written down. It's mostly in the (programming) language and in small groups of programs."

Mr. Harrington: At Chrysler, there was a full-time librarian; programmers used documentation enough to make it well worth while.

Mr. Castin: In general, documentation is very valuable if it is kept up to date, kept all together where you can find it, and otherwise handled well. But if it deteriorates, people ignore it, and there is a vicious circle; it just keeps on getting worse.

Mr. Covill: "With many programs, the problem is that there is no basic strategy. There isn't any way to document an ill-conceived program that will make it maintainable. Mainteability proceeds out of" good basic strategies for the program.

Mr. Brown: At our company, documentation is very valuable, but it's mostly in the form of "Inter-Office Correspondence (IOC)" and not in one big document.

Question: Where does the formal, written documentation reside?

Mr. Brown: "Only in the form of IOCs. And these are a users' guide, not a programmers' guide."

Question: Would a new person be lost . . .?

Mr. Brown: Probably he would, but . . . there are documentation cards in the program. You don't need formal documentation yet; it is embedded in the code."

Question: How useful is out-of-date documentation?

Mr. Hudson: "Out-of-date documentation does have some value, because it may tell the interfaces with system functions." It does not give current details, "but I wouldn't believe them anyway."

Mr. Sleeper: Out-of-date documentation "is very dangerous."

Mr. Harrington: Out-of-date documentation "has a negative impact."

As an idea, you might let the computer do the dog work of documentation. But it would be hard to inspect and do QA on changes if a programmer could enter them directly, without going through an inspector.

(Our Comment: Perhaps you could have a manual buffer in the system. A computer would do the dog work, QA would inspect the changes, and then they could be entered.)

Question: . . . the value of the source coding and comments therein?

Mr. Hudson: Not only at Jacobi but elsewhere, the comments were "extremely beneficial." In spite of having been "misled quite a few times, I've subconsciously developed a reliance on comments. Invariably I will scan the comments first" for clues to what the problem is, and to identify the segment of code which should be examined first.

In spite of relying heavily on them, "I don't write comments. But . . . I will write block comments. I tend to believe them more."

Mr. Sleeper: Very valuable. "We try to take advantage of comments" as much as possible, both in development and maintenance. This "is very neat, because of the programming languages (primarily ALGOL and COBOL in-house, and FORTRAN and BASIC by users) we use."

Mr. Harrington: Source coding is very valuable; comments, less so.

Mr. Castin: Most programmers at SDC would agree with Mr. Hudson. They tend to scan through the comments, looking for key words.

(Our Comment: Perhaps we should look into this idea: Store the comments separately from the code, but put in connectors. Then:

(1) Give the programmer something like KWIC, to help him scan.
(2) Automatically check to see where comment changes are needed when a patch is put in.
(3) Report patches to the right people via the SDI idea.)

Question: . . . the value of . . .
a. Formal meetings with other programmers and systems analysts
b. Informal discussions with others - is it easy to get help from an "expert"?

Mr. Hudson: Fairly useful. Almost all meetings were informal, because the staff was so small.

"There were no experts."

Mr. Sleeper: "It's mostly informal. Formal meetings are usually just a follow-up, to sanctify what has already been decided. We like to keep the group small enough so it will work." The information is very valuable.

Mr. Brown: Regarding formal meetings, "I'm the only liaison with Houston." To the people there, formal meetings are very valuable. They are also very valuable to a local programmer whom Mr. Brown supervises.

Informal meetings are also very valuable; it is easy to talk to Mr. Brown.

Mr. Covill: The most valuable source of information is informal meetings with other programmers. The second is the listings (in COBOL).

Question: To what degree is interchange of information between specialists formalized?
i. What kinds of aids to interchange are there? How well do they work?
ii. What barriers do you observe to interchange of information?

Mr. Harrington: Documentation standards were the formal means of communication.

i. A quality control group checked each document before letting a programmer sign off on it. This worked pretty well.

ii. Personality barriers. Some programmers had to talk, others had to work alone. "Talkers" were assigned to work together, if possible.

Question: . . . the value of the information you carry around in your head?

Mr. Hudson: This was valuable in combination with "analogies with similar problems (on other systems) in the past."

After a time, (experience was) the chief source, plus "what I could dig out of the heads of my co-workers."

Mr. Castin: It is extremely valuable, because you do think by analogy (with similar problems and systems) to a great extent.

Mr. Sleeper: "For those (programs) I have specific responsibility for it's very valuable. But even there it's easy to lose the gory details if you're away for (as short a time as) a couple of months."

Mr. Harrington: So-so, for the non-elite (programming group) but very valuable for "Customer Service."

(Our Comment: An important general rule is that the more urgent requests demand more human memory.

There was a consensus among other programmers at SDC, which was surprising to us, regarding the significant extent to which maintenance experience could be generalized from one software system to a different one.)

Question: What parts of the documentation are you most likely to actually use? How useful is each of the following:

i. Flow charts?

ii. Narrative descriptive material?

iii. Commentary in the source listings?

What parts are you least likely to use?

Mr. Covill: "I don't bother with narrative material. I hunt for what might relate to my problem."

Commentary is good if it is up to date. But "people patch the operative (sic), but not the comments. They can be a booby-trap."

At Chrysler, in COBOL, "the biggest thing of all is to have good record lay-outs."

Mr. Hudson: Flow charts are the most valuable "in the initial exposure. They are more useful in adding capabilities than they are in de-bugging."

Commentary is very valuable.

After initial study, IO specifications are most likely to be used.

Mr. Sleeper: We prefer the documentation to be in the source code; it is most useful there.

"There is another type of documentation which you did not list; the 'interface specifications' . . . which I prefer to use" rather than the more detailed, conventional documentation.

Mr. Harrington: The programmer is least likely to use the narratives, because most are bad.

But the narrative can be very valuable, when well done . . . can be much more concise than flow charts. But analysts vary widely in their ability to write . . . Most don't write well; they leave out essential points, they say things which are ambiguous. Narratives are usually even "worse than manuals." Also, experience with a technical writing course indicates that "you can't make a good writer by legislation."

The commentary in the source listing is "invaluable." Also, it is easier to train people to perform well here.

Flow charts are very useful. Detailed flow charts are most valuable. The place where computer-generated flow charts would be most useful would be in keeping documentation

current, since you could let the computer do the updating. The disadvantages to computer-generated flow charts are that they are hard to read, and poorly organized.

(Our Comment: An important area of research might be in methods of improving the readability of computer-generated flow charts.)

Question: What level of flow chart do you prefer to work with?

Mr. Covill: With higher-order languages, "the low-level of flow chart is not worth the trouble." The test to work with is "the one that gives you the basic strategy of the program."

Covill now works like this: "I draw a flow chart and make my most important decisions. Then I do just one page of code that sort of gives me the feel of what this thing is doing. That gives me the specifications for the main internal workings." In short, "I work from the outside in."

(Our Comment: That is similar to the Mills approach.)

Covill strongly dislikes "any flow chart which is not complete on one 8-1/2 X 11 paper." The flow chart can be layered, but one page has to be a box in another flow chart.

"If you can't show the relationships on one sheet, then you've probably done a bad job of arranging those relationships."

Mr. Harrington: Detailed flow charts were the most valuable to the programmers at Chrysler.

Mr. Sleeper: "I don't think flow charts are too useful, except at the top level." However, it would be good to be able to "call out data-processing functions, like SORT, REPORT, etc. . . . work with big pieces."

## PART III

## MAINTENANCE TOOLS AND TESTS

Question (Introduction): Another area of great interest to us beyond the sources of information is the kinds of mechanized debugging tools available to you.

Question: Do you have a tool which accepts as input a group of programs and turns out a cross index of variables (etc) vs program showing for each program whether the variable is set, used, both or cleared therein?

How useful is it (if you have it), and why? Or, how useful would it be?

Mr. Covill: "The most important thing I've got is a COBOL cross reference . . ." which lists identifier by identifier, and tells where each is defined, and where used.

"Second. The 5500s are all on source language. I can go to it, and not to memory maps and the like." This saves time.

Mr. Hudson: There was just a tool which gave references, i.e., whether a variable was either set or used. But it also could tell what line it was defined on . . .

It was extensively used. Its main drawback was the absence of a set/use breakdown.

Mr. Sleeper: The question is mostly not applicable, because there is very little "inter-program communication on the system." However, one can get "very nice cross-references" on the Command and Edit programs, and such features are very useful, if not essential, in maintenance and improvement.

Mr. Brown: Yes, we have. It is used "quite often." It is "very easy" to use, which is a good point. Its best point is that it "can do other things, and you can use it in league with other documentation aids as a program maintenance tool." It gives a more complete picture of what parameters

are represented in different subroutines. It also generates specification statements required in individual subroutines which use COMMON.

The tool (called COMGEN) is like a set-use program, but it also updates both COMMON and the program, and causes modification cards to be punched.

Question: Would you need COMGEN if you had the INCLUDE of FORTRAN V?

Mr. Brown: "I don't know. The INCLUDE is not being used in the 1108 in Houston, but I don't know why."

(Our Comment: It turned out later that this and other advanced features of FORTRAN V are not used in order to maintain source compatibility with versions of FORTRAN used on other machines.)

Question: Do you have means to cross reference identifiers within a program by line number and showing
where set
where used
from which line numbers it is branched to?

Mr. Sleeper: No, one cannot tell from which line numbers it is branched to. (Would you use such a tool?) "It probably would be valuable."

Mr. Brown: We have "the classical 'set-use' program," but not the line number branch generator.

Question: Does each programmer consider the problem of "fanout"? Is the Problem formally assigned to anyone? To whom?

Mr. Harrington: There was a manual of "inter-relationships and data dependencies." It was generated from the test files. The manual "of course" did not cover all fanout cases. However, new problems were added to the manual as they were reported.

Question: What kind of cross-indexing is there? That is, what tells a programmer that a change at Point A may affect Points P, Q & R?

Mr. Harrington: There is very little cross-indexing between different types of documents, and different levels. But perhaps you could do a lot of that with machine generation of source coding. Also, "why not make an index on which you could use something like KWIC?"

(Comment later: An important research question might be: How do you key each part of each "representation" of the program to each other representation?)

Question: a. Can you get a static memory map?
b. Can you get field explosion diagrams of tables of packed data?
c. Do you have any tools commonly in use which haven't been mentioned?

Mr. Hudson: Yes, there was a static memory map. It was easy to use, and frequently used.

. . . field explosion diagrams? . . . any other tools . . . ?
No, on both counts.

Mr. Sleeper: A static memory map would be "impossible because the system is so dynamic. Only a small part of a program is fixed in a core location, and that part is there only on one execution." But the compiler will "put out a relative map." This is "absolutely essential, to understand where segments are in the dump."

Field explosion diagrams of tables "would be valuable."

(Other tools?) No. "We have nothing to build data sets with, all over the place," and might be a little overwhelmed if these were available.

Mr. Brown: We can get a "Load Map . . . some other kind, but I have never used it."

For field explosion diagrams of . . . packed data, such a tool is incorporated in PACE.

. . . No other tools except PACE and COMGEN.

Question: Can you make symbolic corrections to object code, i.e., patch at a symbolic level without the need to recompile whole symbolic program?

Mr. Castin: The frequency of corrections would determine whether or not you would want to do this in the first place.

Mr. Hudson: No. If it had been available, it "wouldn't have been too valuable."

Mr. Sleeper: "We've supposedly just added a capability to link-edit precompiled program pieces," and this "should help a lot." However, this capability does not seem as broad as that referred to in the question, because all of the definition of the "program piece" is left up to the programmer.

Programmers are not allowed to patch the object code.

Mr. Brown: No. "I would not use such a system. But I might (on second thought) if, on a big system, compile time was scarce." Also, in FORTRAN, (but not JOVIAL and SPL) you can "independently compile subroutines rather than the whole big glop."

Mr. Brown is "very interested in 'segmented compilers.'"

Question: Do you have the means for automatically generating test data for your testing procedures?

Mr. Hudson: No. But it would have been valuable.

Mr. Sleeper: No. (Would you like to have this capability?) Yes, if the formatting, etc. was easy enough.

Mr. Brown: No. But it might be good.

Mr. Harrington: Yes, "there are all kinds of aids. Several levels of test files . . . You can call these with control instructions." They are semi-automatic.

On small sections, you make manual checks. "There are some automated audit programs for massive system tests."

As a "Buck Rogers" idea, it would be good to have "as much machine generation of the clerical material as possible." It would be very desirable, but very hard to have a machine "check the consistency of logic."

(Our Comment: It probably would be possible to have a computer do this, using Boolean logic equations.)

Question: Who sets the test standards?

Mr. Harrington: At Chrysler, the System Programming Staff, who generate the test decks, tapes, and files. Systems Programming sets the standards. Quality Control enforces the standards.

The programmer "gets audited at the end of each phase . . . like . . . design, flow-charting, coding, debugging, check-out, production check-out . . ." The test environment includes samples of actual data. There is a final review of all phases before QC signs off.

Also (this is what we called the Warranty Period Concept), a number N is assigned to each program; the program then has to be used N times in actual production, before the programmer is no longer responsible for it.

Question: What sorts of things slip past the testing procedure?

Mr. Harrington: "Odd combinations . . . seasonal combinations . . . of data. Changes in the structure of the variables."

Question: Does information about the things which slip through feed back and cause any improvement in the test procedure?

Mr. Harrington: Yes. In the "shake-down, it is entered in the log, and integrated" into the procedure.

After N cycles, however, the answer is usually no. Then one usually could not anticipate the things which slip through.

Question: Do you have a selective dump - during and/or post-mortem?
- conditional and/or unconditional?
- choice of formats?

Mr. Huston: Post-mortem and conditional dumps were available. FORTRAN contains a valuable "track-and-trace feature . . . that shows you where you branched from."

(Our Comment: Mr. Hudson is referring to an object time package which works in conjunction with the method used to compile subroutine linkages that provides a "Walkback" listing of the subroutine nesting in effect (with statement numbers of the calls) at the time of an abnormal termination.)

Mr. Sleeper: Except for some "very nice monitor and snapshot facilities in most languages," the answer is no. Would you like it? No, in this facility "the dump is too elusive. There is nothing more frustrating than to get a core dump where the part you want is 'OUT ON DISC.'"

Mr. Brown: Yes, "we have some nice 'revive-execute' sorts of things." Yes, "you have a choice of formats" for dumps.

Question: Do you find that you normally use a lot of PRINT statements while you are de-bugging?

What factors cause you to rely more on one (PRINT, dump) vs the other?

Mr. Hudson: Computer time (i.e., non-prime time) was essentially free to the staff members, and they "usually got a full memory dump, and threw all of it away except three or four pages."

Mr. Sleeper: "Several people do use a lot of PRINT statements; it depends on the individual." Less experienced programmers usually find it easier to use MONITOR and TRACE routines. In BASIC, however, these are not available, so "people have to use PRINT. This is not terribly effective."

More experienced programmers use MONITOR and TRACE less often, because they know more precisely what they want to look for.

Question: Do you have any other display tools which we haven't mentioned?

Mr. Hudson: No, but it would have been good to have ". . . something which would automatically sense some abnormality," and print then and only then.

Mr. Brown: No, but it would be good to have "flow charts which cover all possibilities . . . not limited by" (the ommissions of) the systems analyst.

Question: Aside from the information you've just given us, we'd like to know what things get in the way of your actually using these tools.

Examples:
1. People change the tools without telling you.
2. Operations personnel don't follow instructions, or otherwise goof.
3. User's manual: incomplete; too complex or time-consuming.
4. Temptation to invent one's own tools.
5. Proprietary considerations.

Mr. Castin: Tools are changed before documentation is updated.

Mr. Covill: "Nobody ever seems to include the need for maintenance" in the budgets for personnel, computer time, and equipment.

Mr. Hudson: The main problem was the fact that the "users' manuals were incomplete and inaccurate."

Certain features, such as a decimal register dump, did not actually get in the way, but were irrelevant to the work.

Mr. Sleeper: "We're in comparatively excellent shape here." But in most large data centers, "the almost civil service attitude" gets in the way. "You have no support, and part of that is negative."

A study at Lockheed a few years ago indicated that the probability of a run getting

successfully through all of the steps in the center was less than 0.25.

Question: How about the temptation to invent your own tools?

Mr. Hudson: In general, it is true that "you don't learn what's there; you do your own thing." But at Jacobi the temptation was "minimal, because of the lack of time."

Mr. Brown: "It's more pleasant to invent your own tool . . . You get status. That's the biggest one problem."

Question: Are there any aids which help the programmer convert the static information into an understanding of the dynamics of the program? . . . are any conceivable?

Mr. Harrington: There are no such current aids. "I used a kind of diagram which restated the action . . . know of no work in this area . . . It's up to the individual."

## PART IV

## FACILITIES AND LANGUAGES

Question: What about higher-order languages?

Mr. Harrington: The programmer "can accept what he gets, or try to think like a computer thinks." Recommendation: The programmer should have the easy capability to enter the lower-level language, "not just in a subroutinized way . . . but . . . get a dump of what happened at the machine language level when the statement was executed." Also, one might wish the object code for any statement or group of statements.

(Our Comment: An important philosophical question is: Should software be designed to reward poor programming? Also, this grates against the notion of an optimizing compiler.)

Mr. Harrington: "There is a need for a meta-language that would translate between the application and the language." This would be analagous to the language used in writing a compiler.

You should have debugging routines, not just computational routines, in things like FORTRAN. There are trace routines, but . . .

Accounts Payable was a very large system at Chrysler due to the very large number of vendors. Re-coding Accounts Payable, from COBOL to something like BAL, reduced the run time from 72 to 28 hours. Besides (obviously) lowering cost, this reduction facilitated maintenance, because:

a. It created the effect of making more machine time available.
b. A full system test was made feasible.
c. Re-assembling, following changes, was faster than recompiling.
d. It required different levels of skills and programmer types.

Multiple cross-referencing (e.g., by vendor, part number, length of time since invoice

received, etc.) was a big problem. The files were not big enough to hold all of these data. A higher-order language tailored specifically to Accounts Payable (e.g., permitting statements like DO VENDOR EXTRACT) would have been helpful.

Question: Do you think there is any danger that we will run into English-language-type ambiguities in higher-level languages?

Mr. Covill: "We already have!" It is not generally recognized that "The compiler takes things in a standard order" and has other implicit decision-making features, so that "people don't know that their statements were ambiguous."

But this may be a necessary situation. Otherwise "you are like the old English writers who expressed their thoughts with increasing precision to a decreasing number of people."

Mr. Sleeper: "Yes, there is not only a danger, but you will. But in my opinion, the added 'sensibility' that you get . . . outweighs the danger."

Question: Now a question about Turn-Around Time and over-all availability of machine time.

a. What kind of TAT do you think would be ideal (1) for simple programs; (2) for complex programs?
b. Speaking of your own work habits only, can you envision any problems from a TAT that was very short?

Mr. Covill: Typical turn-around time is one hour, but "brief turn-around times would do the most for me" in debugging. There would be no problem with very short times: "I'd like to have a remote job entry terminal, throw in a job, wait to hear the printer . . ."

Mr. Hudson: Overnight turn-around time was long enough to degrade performance. A four-hour TAT was "about right, except for simple clerical goofs." A very short TAT would "discourage analysis."

Mr. Sleeper: "I get bent out of shape if it takes longer than one hour" for any program. For many, the ideal time would be a few minutes.

(Any problems from a TAT that was very short?) "No!!"

Mr. Brown: For simple programs, a fast TAT. For complex, longer (slower).

"We get immediate turn-around time on time sharing. On batch, . . . two or three hours."

(Any problems from a very short TAT?) "Not knowing what to do, I'd throw in another run, instead of thinking for myself. I'd be less of a thinking programmer."

Question: When you have slow TATs, do you find it hard to work on more than one program at once? In other words, how many projects can you keep active and outstanding at once?

Mr. Covill: With overnight TAT, "It takes time to shift gears. The number of programs I can keep going is not greater than three." However, it helps if the programs are all part of the same system. Having them in the same language also helps, but having them part of the same system is more important.

Mr. Sleeper: "About three."

Mr. Brown: Only one project.

Mr. Castin: When there is overnight TAT, people seem to drift into an arrangement in which each programmer is working on three different projects.

## PART V

## QUALITIES OF A GOOD MAINTENANCE PROGRAMMER

Question: . . . differences in people?

Mr. Covill: "Some people are (very much) better at debugging."

Question: What makes them better?

Mr. Covill: "Their logic and thinking. It's like science. Some people can iteratively form and test hypotheses" and others can't.

Also, "a good debugger will test his original hypothesis so that a false proof lets him make a new hypothesis."

There are two kinds of bad debuggers: (1) "One makes a great big detailed test of A, and learns that that's not the answer." (2) The other "leaps immediately to a conclusion, and puts in a patch." "The leapers and the plodders are both bad."

"The art is sort of figuring how specific" to make the diagnostic tests."

Question: Is training a factor?

Mr. Covill: "People are not taught to program so that their programs are maintainable."

Question: Why not?

Mr. Covill: They write programs which are "absolutely planar. Nothing is modularized."

Question: Why is that so?

Mr. Covill: "They've never had to debug. I'd like to require every new programmer to spend one year debugging before creating any programs."

Question: Do the qualifications for a good debugger remain valid if he moves to an interactive console? Would a terminal be dangerous or wasteful for a poor programmer?

Mr. Covill: The qualifications for the good debugger would "be even more valid, because there would be less time to change strategies."

"A terminal doesn't do anything, one way or the other, for quality. It just makes it happen faster."

"Debugging is a function of the individual and his approach, and not of his tools."

Question: . . . other research on this topic?

Mr. Covill: One relevant research project is going on in the UCSD Psychology Department. It relates to individual differences in "scratch-pad" memory (in programmers?). The director is Dr. Donald A. Norman.

## PART VI

## OPINIONS ABOUT PERSONNEL AND ADMINISTRATION

Question: Anything . . . about personnel?

Mr. Harrington: Personnel policies . . . are perhaps an underestimated area of importance for investigation.

Question: Do you have responsibility for a specific set of programs within the system? OR
Do your assignments rove all over the system?

Mr. Covill: At Chrysler, originally, a programmer was responsible only for "his programs." Later, however, areas were created in which programmers could specialize. "A system of only programs never works. You've got to take the programs out and make them definable entities" within the framework of a larger system, and then make assignments.

Mr. Hudson: Primarily, responsibility for the executive monitor.

Mr. Sleeper: Yes, he specializes, but "I take calls on anything I know about," including maintenance of the hardware.

Mr. Castin: Mr. Sleeper's answer is "most realistic."

Mr. Brown: The assignments do cover the system.

Mr. Harrington: The systems analyst was assigned to a small group of customers, but programmers themselves did not specialize. On the contrary, there was a formalized program of "cross-training," and another "formalized study of common problems."

(Our Comment: The fact that so much effort was spent against specialization might indicate that specialization was really the course of least resistance.)

Question: In general, was the number of programmers assigned to an area proportional to its seriousness? Should it have been?

Mr. Harrington: No, because of the dynamic nature of the system. Different things were critical at different times, and people had to be transferred to the points of temporary problems.

Question: How are emergencies handled?
Are you on call for handling emergency requests?

Mr. Hudson: Yes. There was a skeleton crew, and this was necessary.

There were basically two tasks: First was the routine operations involved in getting runs that customers paid for; this always took priority. Second was optimization or improvement of the system.

Mr. Sleeper: Yes! The corporation is sufficiently small that emergencies can usually be handled informally.

Question: Do more urgent requests short-cut part of the assignment system?

Mr. Harrington: Yes. For example, the car ordering system was tied to the assembly line, and was most vital. For a problem with such a system, the most senior men were on call; they would solve such a problem, or know whom to call. (They were called "Customer Service" for political purposes. Also, their on-call assignments were rotated.)

Question: Now we want to know a little about the kinds of skills your job requires. In particular, do you have to spend very much time in routine "dog work"? (e.g., clerical work, versus thinking and searching for bugs).

Mr. Covill: "There is way too much. For every little change, there is a release memo . . ."

Mr Hudson: "I never viewed the work as dog work."

Mr. Sleeper: Most programmers feel they do, "but I don't." ". . . could use a little (more clerical help), but having a terminal in your office helps eliminate dog work."

Mr. Brown: Yes, programmers complain of the requirement "to make the listing mean a lot more than it normally means."

Mr. Harrington: The use of para-professionals for clerical work should be investigated as a promising idea.

Question: Is there anything in your work that you could classify as physical inconvenience?

Mr. Covill: "Being in a bull pen. You need a little work table for every two or three people. You need tables and blackboards; desks aren't enough."

Mr. Castin: People should change offices when they change functions.

Mr. Hudson: Cross-indexing with the present big stacks of paper is indeed inconvenient.

Mr. Sleeper: "No, just the funny hours."

Mr. Brown: Yes. Simply the physical limits on sizes of sheets, etc., represent an inconvenience.

Mr. Harrington: Importance of the physical environment may be under-estimated. At Chrysler, taking programmers out of a bull-pen improved their productivity significantly. (They were placed in two-man cubicles.)

Providing a small conference room for each four cubicles (i.e., for each eight men) facilitated useful, informal consultation.

Question: Now a question about whom you have to deal with and any problems that arise. Do you get a job request from a customer directly, or does it come through some kind of interface? How well do you think the interface works? Is it intelligible the first time, or does it take several go-arounds? In short, do you have problems interfacing with the customer?

Mr. Covill: No.

Mr. Hudson: In general, the request came directly from the customer.

Mr. Hudson set his own priorities.

The request was "made intelligible in the first session with the customer."

Mr. Sleeper: A typical request for an improvement in the executive system will come from a salesman. It does take several go-rounds before anything comes (if ever) of such a request.

Mr. Harrington: For system modifications, there are two general origins: (1) Users of the system . . . one of 45 groups within Chrysler. (2) The programming staff . . . which usually makes a technical request for something which will improve the usability of the system in operation. Requests for error corrections come from the staff, and may represent emergencies.

Each system analyst was assigned to a certain group of customers. When a request came in,
the systems analyst prepared the top-level flow charts,
the programming supervisor estimated times and schedules and the programmer got a
i. narrative description,
ii. flow chart
iii. IO format (of the user's requirements).

The interface arrangement at Chrysler "was the best I'd seen."

## PART VII

## THOUGHTS FOR THE FUTURE

Question: Now we'd like to turn to an entirely different way of looking at tools for program maintenance. If money were virtually no object, what sorts of tools would you ask to have designed, to make your job as easy as possible? Here are some examples, but don't let them seem to exhaust the possibilities. Suggest anything.

Interactive consoles
Interactive consoles integrated with documentation
Redesign of programming languages to facilitate ease of maintenance
Computer output goes to microfilm and a console oriented system can retrieve and display it.

Mr. Covill: First, "a big old engine that looked like a tap drive, where I could just dial in the kind of error" (e.g., a line transmission failure) "I wanted."

Second, "a system that stored all the current data base definitions, with the OK names for them . . . and a terminal to look at them."

Third, a way of "automatically updating the use table, by version of the system."

Finally, "some way to integrate this whole thing with Operations, so you would know if there really was a program problem." For example, formats which are convenient for programmers can be inconvenient for keypunch operators, who then make mistakes.

Mr. Hudson: "No tools will ever remove having a person with a certain level of talent . . . Programming is partly an art."

It would be good to have "tools to screen out unnecessary information in a dump. If you were a little better at this, you could easily speed up de-bugging by a factor of three."

On big systems, there is a need for a way of taking a correction and "automatically integrating it into the system." This is true because of the (present) high probability of human error.

Mr. Sleeper: "There's nothing I really strongly desired. I'd like an improved set/use program" with very convenient formats and controls.

A minor complaint: Teletype keyboards are awkward to use. Keyboards need to be made more compatible with people.

Mr. Brown: "I received a proposal for 'computer-aided program development,' where you would sit at an interactive terminal, draw a flow chart, and your flow chart would get compiled: You want a flow chart compiler."

(Our Comment: On the console with the flow chart, you could have the computer "mark the lines heavy on the paths that you have used.")

Mr. Harrington: I'd like a console whose buttons would give selective dumps of anything from one word to the whole thing.

"A trace routine coming at me on a console, telling me where I'm at."

On output, not only the output itself, but the status of buffers, through time, which would tell how "this garbage" originated. Now "the programmer has to simulate a computer," which is a bad situation.

Something like a pre-set stop, or break point . . . the sort of thing incorporated in the hardware for machine-language programming of the old computers. (Later note: . . . and something like the old "single-step" versus "continuous" modes of operation?)

Mr. Castin: Here are some things which maintenance programmers at SDC have suggested:

1. A trace which showed only when jumps occur other than to the next sequential instruction. ("It takes so long to look through traces now.")

   Or, trace which would show only when particular items or thin film registers were set.

2. A handy aid would be a display which appeared each time a piece of code was operated or an item was set.

3. A model of the system which would allow input to be tested for completeness of code (prevent fanout).

4. A tool for on-line dumps without using the utility system.

   Or, have the resident utility always in core . . . maybe in an untouchable area.

5. Update documentation by using the computer. Make a documentation change just by changing cards, as with a program.

6. During the system development phase, as the Part I specifications are converted to a set of programs, a series of notes are usually written to communicate inputs to various tables, etc. These should become standard system documents. In other words, the "Implication Notes" should be incorporated in the formal documentation.

7. The ability to parallel run the system with an old and new compilation of a program to point out differences . . . like an experimental and a control group.

8. Centralize all documentation so that each change can be readily seen by the next user. This would also make it easier to keep the documentation up to date.

9. When developing a system, gear the utility tools towards aiding the system in its

development as well as in its maintenance. Or, write the utility programs first, rather than in parallel with the system.

10. A code analyzer which would verify that your patch is not going to adversely affect existing code, branches, and item settings . . . guard against unexpected transfers to your patch.

11. A display of all areas of core your program affects each time it operates. This would turn up implicit references.

12. A trace-back capability where a table of the last file I/O requests, program interrupts, etc. is maintained.

(Our Comment: This 12th request is for a sort of computer analogy of the human's "immediate memory," holding a temporary record of all of the N most recent events.)

Question: Finally, is there anything we should have asked you but didn't?

Mr. Sleeper: None, except to emphasize that "I am totally in favor of higher-level languages."

"There will not be any single higher-level languages as a panacea . . . each . . . will be for its own problem area."

"You can write system software in a higher-level language." Most useful would be something like a version of ALGOL through which a program structure could be implied.

Mr. Brown: "Maybe, 'what makes me the maddest about software development?' What are the frustrations and agonies . . .?"

Mr. Hudson: "You ought to categorize those things the human is going to have to get on board and interact with" in order to decide realistically what tasks can be turned over to the tools.

# APPENDIX III

## BUIC Case Studies and Diary

The BUIC Case Studies which appear in this appendix were developed by Mr. Michael J. Castin of the Systems Development Corporation. The BUIC Diary was kept by Mr. Tom Brotherton also of SDC. The commentary on each diary entry (generally below the solid line) was supplied by Mr. Castin.

A Glossary of BUIC Terminology appears after the Diary together with explanatory material on the BUIC Error Correction and Production Cycles.

Brief résumés of Mr. Castin and Mr. Brotherton appear at the end of this appendix.

# BUIC CASE STUDIES

## BUIC CASE STUDY

TITLE: Poor Documentation

PROBLEM DESCRIPTION: In order for a maintenance programmer to perform at maximum efficiency, he must be able to install program changes quickly and effectively. To accomplish this, he must be able to determine quickly the affected areas of the program. If programs are not well documented, the programmer will be considerably slowed down.

BUIC EXAMPLE: Recently, a change to one of the BUIC Manual Input card formats was requested by ADC. The programmer assigned responsibility for the Manual Input function estimated the time required to install the necessary code. He had been assigned to that area only two months earlier but felt that the change was not complex to install. Problems arose when he discovered that the existing code had already undergone a great deal of modification and the documentation described the area only in overall terms. This made it very difficult to determine the logic flow through the affected program area.

The programmer finally delivered the code by spending 120 hours on developing the change rather than the 60 hours he originally estimated. He delivered it only 3 days late by working overtime and on his vacation time.

## BUIC CASE STUDY

TITLE: Uninformed Third Party

PROBLEM DESCRIPTION: In a complex large system, changes to one area may affect other areas. If the change coordinator fails to inform all affected parties, incompatibilities and schedule slippage may occur.

BUIC EXAMPLE: The addition of a Real Time Quality Control function to BUIC necessitated the addition of a new program module to the Air Defense Program (ADP). Adding that module caused the operating sequence of existing modules to change. The coordinator of the new product was not aware that a special module timing processor required modification whenever program sequencing changed.

As soon as the new module was loaded on the ADP master tape, the programmer responsible for analyzing module operating time began to experience difficulties with the timing processor. He estimates that 4 days were spent in determining the cause of his difficulties.

Although no schedule slippage occurred as a result of this problem, 4 days which could have been put to better use were wasted.

## BUIC CASE STUDY

TITLE: Language Requirements

PROBLEM DESCRIPTION: The language in which a program is coded will limit the logic available to the programmer. Additionally, he must code in the techniques the assembler/compiler will accept rather than the techniques he might otherwise choose.

BUIC EXAMPLE: Whenever new programmers are transferred into the BUIC project, they must learn not only the functional requirements of their area but the language requirements of the JOVIAL compiler. In reality, this is not confining since the compiler contains the attributes required by BUIC's design.

A good example of unique language requirements is that of the capability of the BUIC compiler to deal with individual bits of a data word. Not all compilers have this feature and programmers must learn how to use it when they join BUIC or how to get around it when they leave.

## BUIC CASE STUDY

TITLE: Indirect Addressing

PROBLEM DESCRIPTION: In BUIC all indirect addresses are relative to the start of the applicable programs data region (BAR). The BAR is located right after the instruction region and each time instructions are added the address of the BAR is moved down accordingly. In order to allow this dynamic updating an indirect address list is created for each program and put in a specific location so that it can be updated with each change.

Unfortunately one cannot add to this list after a program is compiled. (SRC will not create a new entry.) When programmers attempt to use a new indirect address it will work only until the location of the BAR moves.

BUIC EXAMPLE: In a previous version, a corrector was issued to the field sites which had an indirect address reference which was not in the indirect address list. The corrector worked fine until somebody ran a test which changed the affected programs BAR. The system hung up and a new corrector had to be written which did not contain a static indirect address.

## BUIC CASE STUDY

TITLE: Modifying Table/Item Locations

PROBLEM DESCRIPTION: Whenever the location of a table or an item changes, all references to the table or item must be changed accordingly. In a large system it can be quite difficult to discover all references.

BUIC EXAMPLE: When table and item locations change in the BUIC system, all references to the items and tables must be recorded using either the compiler or Symbolic Relative Corrector. The only tools for discovering all references are the SET/USE listing and the Tag Reference listing.

Problems have arisen in BUIC because of the following limitations of the two listings:

1. They are accurate only to the most recent compilation of each program.

2. Implicit references such as those used by Pseudo Instructions are not detected by either tool. (An example of a Pseudo Instruction would be the CYCle Instruction which will shift a register from 1 to 48 bits dependent upon item size and location.

## BUIC CASE STUDY

TITLE: Hardware Limitations

PROBLEM DESCRIPTION: Hardware limitations and idiosyncrasies can cause errors and problems not readily visible in program listings.

BUIC EXAMPLE: In BUIC the capability exists to modify the next program address to be executed. This allows an internal subroutine capability. A problem can occur, however, when a multiply or divide instruction is executed just prior to one of these internal subroutine jumps. The problem is called double overlap fill and can cause an incorrect program address to be selected. Although this hardware limitation is documented, occasionally a programmer either neglects to read all available documentation or just forgets that the problem exists.

During the development of the previous BUIC version, one of the program modules schedules to be recompiled was not available to be loaded on the master tape on the planned date. The programmer in charge of the program recompilation complained of recurrent program halts in an area that he swore was error free. A senior programmer was assigned to assist him and after a week discovered the source of the problem.

Earlier in the program, a double overlap fill was occurring just before an internal subroutine jump. The result of this was a return to an incorrect address after the subroutine operated. The program then attempted to operate program data as if it were an instruction and ultimately halted in the routine in question.

BUIC EXAMPLE (cont.): The result of this problem was two man weeks spent in non-productive work and the loss of a week in available time for testing the newly compiled module in a system environment.

## BUIC CASE STUDY

TITLE: Home Office Test Procedures

PROBLEM DESCRIPTION: The test methods practiced in the production shop may differ from those practiced by the user. This can create a situation where the user uncovers an error even after the production shop thoroughly tested its product.

BUIC EXAMPLE: When the BUIC system is cycled up in Santa Monica, a standard start-up card deck is utilized. One of the functions of that deck is to initialize the system as being in the Simulation mode.

One of the previous versions contained an error which was evident only when the system was initially in the Live mode and then changed by switch action to the Simulation mode. This error existed all through our production cycle but was never noticed because of our start-up procedures.

## BUIC CASE STUDY

TITLE: Time Constraints

PROBLEM DESCRIPTION: Because of the large amount of time required for testing each new version of a large system, programmers are constrained by the time allotted for producing new products. They are further constrained by time requirements for familiarization, coordination, documentation, other work, and their own testing needs.

BUIC EXAMPLE: The BUIC Guidance function is difficult to check out completely because of the infinite combinations of Interceptor Position, Speed, Heading, Tactic, Altitude, etc.; Target Position Speed, Heading, Altitude, etc.; and Cycle Time, Track Load, or other system influences. Given a situation such as this, it should be obvious that to thoroughly test the Guidance function would require an extraordinarily large amount of time.

Since there must be a cutoff time for testing, it is impossible to uncover every error which may exist in the Guidance function. Because of this, errors are continually found in the Guidance program module even though the caliber of the programmer maintaining the area is usually above average.

## BUIC CASE STUDY

TITLE: Data Reduction Time Lag

PROBLEM DESCRIPTION: When the master tape for a large system is being continuously updated, some sort of overall complex system test should be run periodically to insure continuous quality. The amount of data reduction, the time to obtain that reduction, and the amount of time spent analyzing the reduction can get to be quite large.

BUIC EXAMPLE: Approximately once each month during the BUIC production cycle a large system test, the FQT, is run. This test is reproducable and documented; there are predicted outputs for each BUIC function. A complete data reduction run and analysis can take up to 10 days, varying with the number of programmers involved. That is a long time to certify a tape.

Usually, to circumvent the time lag, spot checks and minimal data reduction passes are run. This cuts certification time way down but is not in keeping with the intent of the test: maximizing quality control.

## BUIC CASE STUDY

TITLE: Little Used Support Programs

PROBLEM DESCRIPTION: With a large system there are usually a number of Utility Support programs available to aid in the Maintenance Task. For various reasons some of the programs are used much less than others or not at all. Some of the contributing factors to this lack of usage in BUIC are presented below.

BUIC EXAMPLE: Listed here are some of the BUIC Utility and Support Program/Systems which are used less frequently than others. Accompanying each Program/System is a consensus opinion on the reason for that lack of usage.

Parameter Test Tool - Originally designed for checking an individual program module outside of the system context. Used extensively in the system development phase prior to the completed system. Now that the system is operational it is no longer necessary and is harder to use than system-oriented tools.

Trace - The use of the trace function decreases with the increase of programmer experience. As programmers become more familiar with program design the need for trace disappears. Also one of the trace options is rarely used--a capability for outputting the trace dump directly to the printer instead of using a DLO tape. It takes an exorbitant amount of time.

Load Octals on Tape - Unused because the Symbolic Corrector Loader or Symbolic Relative Corrector functions are far superior.

Dump - A dump option, direct on-line printout is so slow that it is rarely used. The normal procedure is to dump onto a tape and print the tape later.

BUIC EXAMPLE (cont.): Assembler - Only one function of the assembler is even used, the update function, since all programs are compiled with the JOVIAL Compiler. The update function allows the user to update a symbolic prestore tape of a program without rereading the entire symbolic deck.

Dynamite - One function of Dynamite allows a user to set a COMPOOL item to a predetermined value in a specific cycle. This feature is rarely used because the planning of what to set and when to set it takes longer than using a simulation tape to set up the desired environment.

## BUIC CASE STUDY

| | |
|---|---|
| TITLE: | Specialization |
| PROBLEM DESCRIPTION: | Whenever the knowledge of a particular system function is confined to one person (or "expert"), the maintenance programmer needing assistance in that area is restricted by the availability, knowledge, and idiosyncrasies of the "expert." |
| BUIC EXAMPLE: | The BUIC Lateraltell function, communication between two or more defense facilities, is an area which is usually assigned to one programmer. That programmer usually becomes the only person knowledgeable in lateraltell because other programmers tend to be unsure of themselves as communications experts. Consequently, when a product or error has lateraltell implications, the coordinator relies heavily on the lateraltell programmer.<br><br>Whenever the lateraltell programmer is ill, on vacation, or just busy, products requiring his assistance are held up until his availability. In most other functions, programmers are more sure of themselves and they will code the change. |

## BUIC CASE STUDY

TITLE: Programmer Idiosyncrasies

PROBLEM DESCRIPTION: One of the common complaints of maintenance programmers is that a program module was originally written with such sophisticated coding techniques that it is difficult to maintain. Programmers must draw the line between maximizing computer attributes and developing easily maintainable programs.

BUIC EXAMPLE: One of the BUIC program modules, the sort program, was developed by a programmer who relied heavily on utilizing the thin film or stack capabilities of the computer. Most of the other BUIC program modules utilize a combination of thin film coding and temporary data registers to make the code easier to follow.

Currently, whenever a programmer wishes to install a change to the sort program, he must spend three times as long on that program as he would on any other since he must insure that his new code does not disturb any thin film registers already used by the program.

BUIC DIARY

BUIC DIARY: MARCH 8, 1971

-- Spent some time desk checking the newly compiled switch program, KAW.

No incidents.

USED: Tag Reference

Set/Use

Symbolic Relative Corrector Listings

Compool, Comdoc

Program Change Specification - Generated by Programmer

Part II Specification

-- Wrote code for EPD 027.

No incidents.

USED: Program Listings

Tag Reference

Symbolic Relative Corrector Listings

Compool, Comdoc

EPC 027 Document

Part II Specifications

---

Tom spent most of the day at his desk using previously procured printouts from the above tools.

BUIC DIARY: MARCH 9, 1971

-- Desk checked KAW using same materials as previous day.

No incident.

-- Coordinated Group Test

USED: Symbolic Relative Corrector (SRC)

Dynamite

Pre-recording

Dump

Start BUIC

-- Ran EPC 027 Test

USED: Start BUIC

Load (Function)

Symbolic Relative Corrector (SRC)

Experienced problem due to mispunched corrector. Aborted job to be rescheduled.

-- Problem was reported in Bookkeeping program. Spend 10 minutes discovering problem was corrected but was not fixed on present Master Tape. Scheduled for next load.

---

Tom ran the Group Test without incident. The load function, used for EPC 027, was used as a time saver. His SRC deck was so large that he was using approximately 10 minutes of computer time just to read in his card deck.

BUIC DIARY: MARCH 10, 1971

-- Continued to desk check KAW.

Received DOC listing.

No incident.

-- Corrected problem in EPC 027.

Submitted production job to load test master. Deck included:

Symbolic Relative Corrector cards

Load Control cards

Compool Octal cards

-- Worked with another programmer in uncovering and solving problem in Manual Inputs program.

USED: Symbolic Relative Corrector

Pre-recording

BUIC Analysis and Reduction System

The above problem was discovered by other programmer while working on another error.

---

Tom was sure of his EPC 027 fix and felt safe in submitting job for production (to be run at night without his supervision).

The Manual Inputs problem was discovered when the other programmer fixed an outstanding problem in his program. The second problem was not noticeable while the first existed.

BUIC DIARY: MARCH 11, 1971

-- Completed desk check of KAW program - submitted acceptance memo for typing.

Used special processor of BARS.

-- Coordinated Group Test

USED: Symbolic Relative Corrector

Dynamite

Pre-recording

Dump

Print Function

Start BUIC

-- Analyzed suspected incompatibility between two volumes of the BUIC Operational Specifications.

-- Reviewed load of EPC 027 for possible errors.

USED: SRC Printout

-- Wrote documents for 4 error corrections previously tested and submitted for load on the Master Tape.

---

The incompatibility was found to exist in the two documents. A decision on how to correct it was deferred. Tom seems to spend more time documenting than coding, typical of all programmers.

BUIC DIARY: MARCH 12, 1971

-- Reviewed a draft copy of BUIC Program Change 361. A parallel activity to EPC 027 for a later version.

No tools used.

-- Rewrote BPC 358, "Add New Function to IDO Console"

USED: Old BPC

Part I Specification (in using this specification, Tom discovered a word had been left out and reported it to the person responsible for maintaining the document).

Pre-recording

SRC Printouts

-- Discovered loop in Bookkeeping program

Used SRC printouts and program listings to find problem.

---

Tom had to rewrite BPC 358 because of a conflict with another BPC which used the same switch action. (A fanout related problem.)

Tom spent about 15 minutes finding the Bookkeeping loop. An ensuing discussion brought out the fact that the loop was caused by a previously installed correction branching to an octal location rather than a tag. This branch was not evident to the programmer searching the listing.

BUIC DIARY: MARCH 15, 1971

-- Reviewed DOC listings of MIN and BOK to insure their accuracy.

Participated as an observer during tests on the newly loaded ADP master.

Verified that correctors to MIN on that load were installed as anticipated.

USED: SRC Printout

-- Recoded portions of EPC 027 to save spare registers.

Saved 22 registers.

USED: Coding instruction manual.

---

Tom reread the Burroughs Coding Manual and found ways to combine instructions and use more efficient instructions. This allowed him to save the register space in the EPC.

BUIC DIARY: MARCH 16, 1971

-- EPC 027 - Ran test on newly generated code.

-- Uncovered hang-up in Manual Input Program (MIN).

Investigated hang-up.

USED: SRC Printout

MIN Listing

Tried to run trace to pinpoint hang-up but computer problems aborted job.

---

Tom spent about half the day consulting on a Non-BUIC project.

CIRAD Comment: The use of tracing mentioned here and in the following entries seems to contradict the characterization of trace given on Page B-13.

BUIC DIARY: MARCH 17, 1971

-- EPC 027 - Reran trace but a drum problem caused the job to abort.

Spent time looking at SRC printouts and MIN listing and discovered cause of hang-up (Pending Test).

---

Tom left work early today. His wife needed transportation home.

The hang-up was caused by using correctors which did not apply to the current program mod. Tom had installed this EPC in a previous BUIC version and in his attempt to shorten his job, he tried to use as much of the old code as possible. He inadvertently used some code which no longer applied to the MIN program. We normally insert the applicable program mod as a comment on the symbolic cards.

BUIC DIARY: MARCH 18, 1971

-- Spent the morning in preparation for and attending a meeting regarding this diary.

-- Worked on EPC 027 reviewing code using SRC Listing and Program Listings. Computer problems prevented code test.

-- Spent some time writing documents for use by training team in North Bay, Canada.

---

Tom explained some of the production cycle to Levi Carey and responded to questions from Dr. Wersan and Dr. Overton.

The training documents are to be used in teaching the maintenance of various functional areas of the SAGE system to Royal Air Force personnel.

BUIC DIARY: MARCH 19, 1971

-- Continued work on training documents.

-- Set up an EPC 027 computer test utilizing:

Tape Load

Pre-recording

Symbolic Relative Corrector

Start BUIC

Could not cycle and returned to office.

Spent remainder of day with Startover programmer determining cause of cycling problem.

---

I questioned Tom on the possibility of his uncovering new maintenance aids as he produced the training documents. He said that the documents were geared toward learning the area more than learning maintenance techniques.

The BUIC Startover program is responsible, among other functions, of initializing and starting the BUIC Air Defense Program cycling. The error in the Startover program correctors for EPC 027 prevented the BUIC ADP from cycling. There was no way to get around it. It was not discovered previously because Startover correctors must be loaded and cannot be read by SRC.

BUIC DIARY: MARCH 22, 1971

EPC 027 - Tested correctors to solve problem of March 19. Apparently the correctors work, but discovered another error caused by mispunching of the cards coded March 15.

Later in the day retested correctors; they appeared to work okay (within the limits of the test).

USED: Tape Load

Symbolic Relative Corrector

Pre-recording

Dump

Submitted a production job to obtain MDT listings of programs MIN and BOK incorporating all current EPC 027 changes.

---

The secondary problem noted today seems to be a typical one facing the maintenance programmer--that of fixing one problem and either discovering or causing another.

BUIC DIARY: MARCH 23, 1971

-- EPC 027 - Ran test with existing correctors and encountered a loop in program MIN.

USED: SRC

Pre-recording

Dump

Print

Trace

Trace was not helpful in discovering loop because the problem involved a branch outside of MIN's core area and trace brake down when this occurred.

Tom worked on the EPC in his office and discovered his problem by looking over the program listings.

---

My discussion with Tom turned up the cause of his program problem. He had disturbed the positioning of the stack in a routine where the return address of the calling routine was stored in one of the stack levels. This type of problem is typical of programs which rely heavily on stack coding. (Using the stack instead of temporary core storage.)

There is a timing and storage saving by stack coding but the routines are not easily maintained. This was my first experience with trace failing to find the problem.

BUIC DIARY: MARCH 24, 1971

-- EPC 027 - Planned computer test this morning to verify corrections to yesterday's problem. Memory parities caused the job to abort and machine was turned back for maintenance.

Tried again in afternoon and job ran successfully. Loaded a new master tape with all correctors to date.

USED: Symbolic Relative Corrector (SRC)

Pre-Recording

Tape Load

---

Each time Tom has a successful computer run he varies his inputs somewhat. This causes him to go from a successful run to an un-successful run with seemingly no changes in his correctors.

He used Manual Input cards in his testing and this allows him an almost infinite number of variables. Manual Input cards are read in through the card reader while BUIC cycles and allow dynamic change of certain elements in the BUIC environment.

BUIC DIARY: MARCH 25, 1971

-- EPC 027 - Ran a test utilizing the newly loaded Master Tape.

USED: Symbolic Relative Corrector

Pre-recording

Start BUIC

Dump

Trace

The test turned up a problem in the Tabular Display associated with the EPC. The display contained zeroes instead of valid data.

Further investigation showed the problem to be caused by an incorrect constant used by the program. An octal card was added to the EPC to solve this problem.

---

It appears thus far that Tom's main test tool has been to continuously exercise the EPC code, each time varying the input. Perhaps we should have kept track of the number of different paths through this code.

BUIC DIARY: MARCH 26, 1971

-- EPC 027 - Prepared for computer run to test the correction from yesterday. The system could not be cycled and memory parities plagued the run. The computer was turned back to the maintenance people. No further activity on the EPC took place.

-- BPC 358/01 - Worked on the document the remainder of the day.

---

With the exception of EPC 027, activity on producing the new BUIC version has slowed to almost a halt. A pre-release tape has been built for shipment to Fallon on April 2. Shortly thereafter problems will begin to be reported and activity will increase, but for the meantime, Tom's main effort will be concentrated on EPC 027.

During this time period, almost all activity is directed towards producing Version Documentation, Specifications, Users Manuals, and Operator Handbooks.

BUIC DIARY: MARCH 29, 1971

-- EPC 027 - Spent one hour preparing for computer time.

During computer run loaded a new Master Tape and ran test of current corrections.

Encountered one problem when a Manual Input Card is read in to status aircraft at an airbase, another card is read in to clear that status, and the first card is read in again.

USED: Symbolic Relative Corrector

Pre-recording

Trace

BUIC Analysis and Reduction System

Start BUIC

Dynamite

-- Worked on BPC 358/01 documentation.

---

Tom's heavy reliance on computer time for his testing must in some way lend itself to justifying the design of on-line debugging tools.

BUIC DIARY: MARCH 30, 1971

-- EPC 027 - Set up and ran a trace on yesterday's aircraft status problem. The test was delayed due to difficulties with tape drives: repeated tape parities prevented cycle-up.

The maintenance people corrected the problem and the trace operated without further incident.

First analysis of the trace did not turn up the cause of the problem.

-- BPC 354 - Researched this BUIC Analysis and Reduction System program change in preparation for conducting a test on its accuracy.

---

Tom is still studying the trace in an attempt to discover the program problem. The problem is apparently not readily evident.

It will be interesting to see what type of problem was not obvious upon first studying the trace.

Tom is acting as the "naive" third party in testing BPC 354 as he did with the testing of the KAW program.

BUIC DIARY: MARCH 31, 1971

-- EPC 027 - Ran another trace in order to develop more information about latest problem.

Problem was finally discovered by analyzing a combination of trace output and program listings.

---

The error turned out to be a mistake in program design logic. This was not as apparent as an incorrect branch or a setting of the wrong item. Analysis of the trace initially showed everything as working correctly because Tom was investigating it from a standpoint of program error rather than incorrect logic.

# BUIC GLOSSARY

# and

# BACKGROUND INFORMATION

SET/USE

This is a program which produces a matrix cross-indexing BUIC programs and BUIC compool items. At each intersection, a symbol indicates whether a program sets, uses, sets and uses, or clears an item.

INDIRECT ADDRESS AND BAR TABLE REFERENCE

This program lists each table, its type, its address, and its length, accessed by each program. It also produces a picture (Field Explosion Diagram) of each compool table structure.

PARAMETER TEST TOOL

A program which allows the programmer to test his program without loading it on the master tape. Parameters are set up as input to his program and the output is saved for analysis.

FACILITY SYSTEM

A subset of Utility programs which are duplicated on the ADP master allowing tape loads, symbolic correction, dynamite, and recording without reading the Utility master.

TAG REFERENCE

A program which produces a listing for your program indicating where it accesses items, tables, internal data words, internal program tags, and all thin film references except the stack.

SYMBOLIC CORRECTOR LOADER

This allows you to input a binary tape of a program, add symbolic correctors to that program, and end up with a binary tape of the corrected program.

COMPUTER UTILITY AND SUPPORT SYSTEM EXECUTIVE

This is the Utility control program. It reads input cards and based on the control information on those cards, it branches control to the applicable Utility program.

DUMP FUNCTION

This produces, on tape or printer, an octal dump of any requested area of memory and/or a dump of the contents of the thin film registers.

BUIC Error Correction Cycle

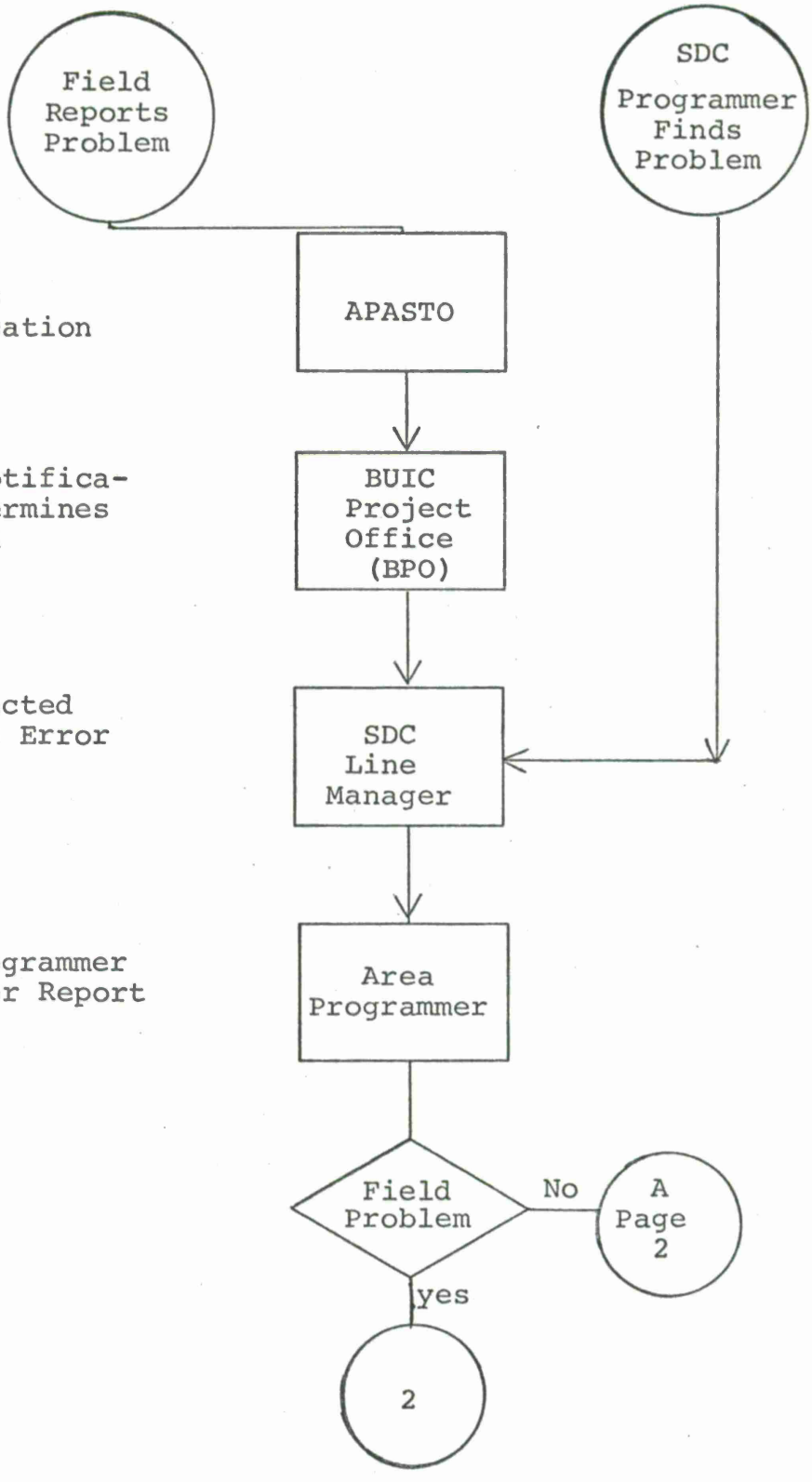
1. Problem Uncovered

2. APASTO Receives Error Notification

3. BPO Receives Notification and Determines Area Impacted

4. Manager of Affected Area Receives Error Notice

5. Responsible Programmer Receives Error Report



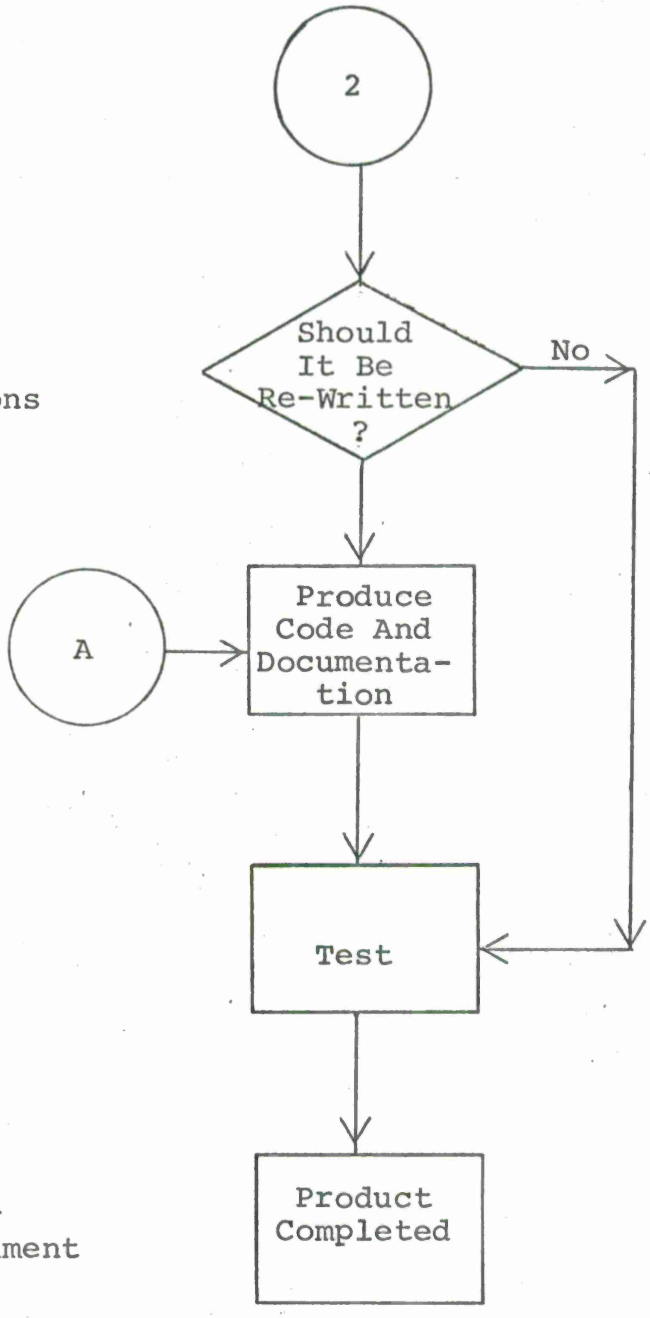
BUIC Error Correction Cycle

6. Field Problems Sometimes Have Military Solutions

7. Programmer Writes And Documents Code

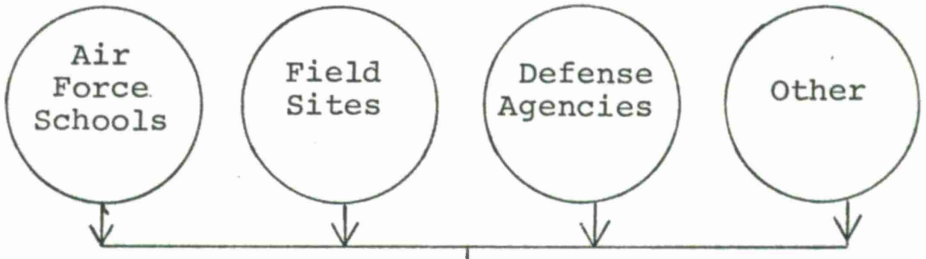
8. Coded Change Is Re-Verified To Prove Error No Longer Exists

9. Error Correction Loaded On Master Tape And/Or Document Sent to Field



BUIC Production Cycle

1. Design Change Suggestions (DCS) Submitted




2. ADC Approves

HQ. Air Defense Command (ADC)

3. APASTO SDC Office Receives Memo

APASTO

4. BPO Receives DCS And Determines Function(s) of BUIC Affected

BUIC Project Office (BPO)

5. Manager of Affected Area Receives Work Request

SDC Line Manager

6. Manager Assigns Programmer/Designer As Product Coordinator

Product Coordinator

1


7. Coordinator Calls Meeting Of All Affected Parties

8. Minutes of ATSM Are Approved By APASTO And Published

9. Changes to System Design Are Published

10. A Draft Document Explaining All Anticipated Program Changes Is Produced

11. After Manager Approves Program Change, Coding Begins

Linear Days (not working days)

14.3

35.9

33.9

9.0

9.0

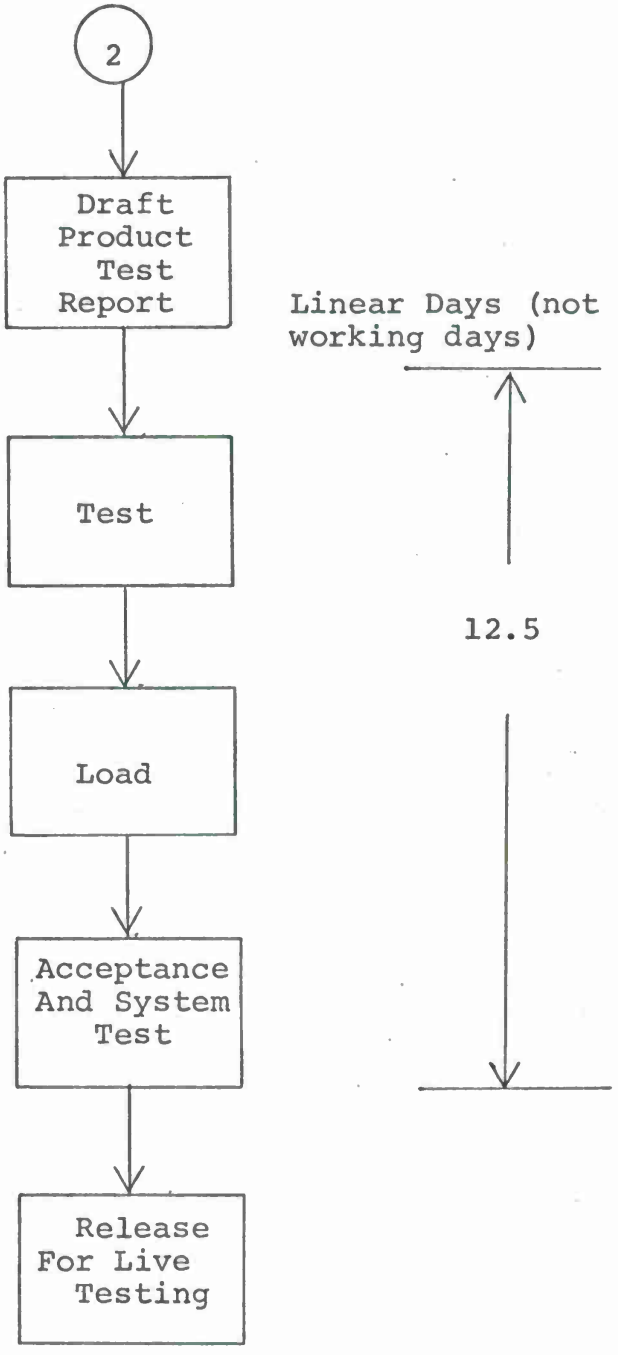
12. Coordinator Submits Anticipated Test Plan For Approval By Manager

13. After Test Plan Approval Coordinator Tests Product

14. Product Submitted For Incorporation On Master Tape

15. 3rd Party Verifies Accuracy Of Product And Its Impact On The System

16. Master Tape Released To Air Force For Two Months Live Testing



## ADDITIONAL BUIC PRODUCTION CYCLE EXPLANATIONS

### BOX 7 - CALLING THE ANALYSIS TEAM STUDY MEETING

At this point in the production cycle, the coordinator must determine all areas which will be affected by this change. He then sends meeting notices to each programmer which he feels is involved as well as each line manager responsible for BUIC production. The managers do not normally attend but they send a representative if they think their area will be affected.

The additional check by the managers usually serves to avoid an area being overlooked. Occasionally, however, a manager is very busy and does not notice the meeting invitation until the meeting is over.

### BOX 9 - PUBLISHING THE PRODUCT CHANGE SPECIFICATION

The Product Change Specification contains the changes, additions, or deletions to the BUIC Part I Specifications. In order to insure that the document is complete and accurate, the coordinator routes a draft copy to all BUIC line managers for review. Usually six days is allowed for comments to be returned.

This affords another check to insure that all affected areas are aware of the change. The managers route the specification to their programmers and checks are made for such things as the feasibility of the proposed design, again the possibility of an area being overlooked, or conflict with other proposed changes.

After all comments are received, evaluated, and installed, the final document is typed, approved for incorporation into BUIC, and published.

### BOX 10 - DRAFT PROGRAM CHANGE SPECIFICATION

This document contains all changes anticipated to BUIC program modules and data base to incorporate the design change. The change coordinator collects a prose description of each program module's changes from the assigned programmer and adds any data base changes and/or equipment changes to produce the document.

The draft document is then routed to all line managers who reroute it to their programmers to ensure accuracy and completeness. After all comments are received, the coding is installed

in each module, the document is reviewed again to verify that the actual changes were the anticipated changes, and a final Program Change Specification is produced.

## BOX 12 - DRAFT PRODUCT TEST REPORT

After a Program Change Specification has been approved and coding is underway, the change coordinator produces a draft Product Test Report. This document explains how the coordinator intends to verify that the change performs as specified. The level of testing is such that the new or changed code is verified in a sterile or non-system environment. Further testing is accomplished during the Acceptance Test activity (see the description of Box 15).

The coordinator submits the draft to his line manager who reviews it for technical accuracy, completeness, and responsiveness to the intent of the design change. After testing is completed, the Product Test Report is reviewed to verify that the proposed testing was indeed accomplished and the final document is published.

Perhaps it should be noted here that some system testing does take place to verify that no obvious degradation occurs to the existing Air Defense Program. This is, however, outside of the intent of the Product Test Report.

## BOX 15 - ACCEPTANCE AND SYSTEM TEST

After a coordinator has completed his product and it is loaded on the Air Defense Program (ADP) master tape, it is submitted for acceptance testing. The acceptance test is conducted by a randomly selected third party. He prepares a test plan memo and tests the new product in its system environment.

At the conclusion of his testing, he transmits a memo to the product coordinator indicating acceptance, partial acceptance, or rejection of the loaded product. In either of the last two cases, the coordinator makes the required changes and resubmits the product for load.

Another activity, the system test activity, parallels this effort. During this activity the latest ADP master tape, which includes many new products, is subjected to two successive simulation tests.

## ADDITIONAL BUIC PRODUCTION CYCLE EXPLANATIONS

The first test, the Heavy Load Formal Qualification Test (FQT), verifies the proper operation of the current ADP master in a peaked load environment. Timing data is also produced during this test which is used to determine any variances in module operating times from load to load.

The second test, the Sim Mode FQT, verifies the proper operation of the current ADP master in a "normal" day to day air defense environment. Data reduction from this test is compared with previous test reduction to uncover any new errors. Any errors noted are reported to the responsible programmer and corrected. The Sim Mode FQT is updated periodically to implement new ADP design changes.

## MICHAEL J. CASTIN

Mr. Castin has a Bachelor of Science degree in Business Administration from UCLA.

Mr. Castin is able to apply over seven years experience in Design, Projection and Testing of Computer Program Systems to his present position. Mr. Castin has been involved with Real Time Command and Control Systems, Data Base Conversion, Data Management and Report Generation as well as digital communications between large scale Systems.

During his experience Mr. Castin has been involved with the IBM 7094 and System 360 computers plus the Burroughs D-825 computer system utilizing the FORTRAN, COBOL and JOVIAL higher order languages in addition to direct code machine language applications.

In Mr. Castin's current assignment he has responsibility for the technical coordination of work produced by a section of ten programmers. In fulfilling this assignment he assures correctness and quality of code as well as documentation. Other duties include training, procedure development, and customer liaison.

TOM BROTHERTON

Mr. Brotherton has a Bachelor of Science degree in Physics from U.C. Riverside. He has been a Programmer at SDC since 1968, his first full-time job. He is responsible for maintaining the BUIC Manual Input program (MIN), the BUIC Bookkeeping program (BOK), and the BUIC Automated Programmed Instruction program (API). He was not involved in the production of any of these programs.

His experience includes test design for SAGE and BUIC, and some involvement with the BUIC Data Reduction System. He is familiar with PL/1, COBOL, FORTRAN, JOVIAL, and 360 and Burroughs Assembly Language.

His current assignments include: coordinating EPC 027, maintaining the three programs mentioned, and verifying the correctness of the new mod of the BUIC Weapons Switch Program (KAW). In the latter activity, he is performing as an uninformed third party.

# APPENDIX IV

## A Study of Factors Inhibiting the Effectiveness of Maintenance Programmers at Chrysler Corporation

CONTENTS

CONTENTS

CONTENTS

INTRODUCTION: This Appendix is organized in three major sections. The first section describes, chronologically, the activities associated with the maintenance programming task and associated inhibiting factors. The second section is organized by inhibiting factor and a brief description of the manifestation of each as experienced at the Chrysler Corporation by the author. The third section describes a subjective scoring/ranking system for the evaluation of the effect of inhibiting factors as experienced at Chrysler, and presents such an evaluation.

The contents of this Appendix are based entirely on the experiences of the author while employed at the Chrysler Corporation where he was in charge of a staff of 260, including 100 programmers, who used 21 computers. A large part of the effort involved an on-line order entry system for the control of assembly line production. A more complete author's resume appears at the end of this Appendix.

I. Maintenance Programmer Processes and Environment.

A. Chronology of the Process.

The following exposition is intended to define the processes and functions associated with the maintenance programming task and to describe the inhibiting factors observed to be associated with each.

1. Initial Request for Change and Systems Analysis. A request to change an operating program is inaugurated. The inauguration for such a change may come from two sources: the customer or user of the program's results and the programming staff. If inaugurated by the customer the change usually corresponds to a change in the application requirements or to a desire for a format or presentation of the data that represents an improvement in the ability of the user to utilize the results. If the change is inaugurated by the programming or systems staff it usually represents one that will improve the execution or maintainability of the program. Factors in this process that inhibit programmer effectiveness are:

a. The request for a change is verbal, or scantily documented. This causes the maintenance programmer to have to interact with the customer in an iterative learning process that leads to understanding of the exact requirement for change. Many times, the inefficiency of this process leads to the creation of a Systems Analyst position as a buffer between the user and the maintenance programmer. This, in turn, may inhibit programmer effectiveness in the following way:

b. The systems analyst is either not familiar with the program requirements or with the user requirements, or both. Thus the iterative learning process is complicated by the introduction of another level in the communication process.

c. Even though the Systems Analyst, when present in the organization, may be fully familiar with both user and program requirements, the lack of a fully defined and agreed to documentation language may prove to be an inhibiting factor.

d. During the change request process, lack of knowledge of the program to be altered will inhibit the maintenance programmer's effectiveness. This manifests itself in the programmer's inability to make accurate estimates as to the time and resources required to perform the task requested. Poor estimates lead to later renegotiations with the using authority with an attendant loss of time and efficiency on the part of the programmer.

e. Involvement of the programmer in the change request process frequently takes him away from another in-process maintenance task, thereby delaying that task. This delay in the orderly process of pursuing maintenance tasks introduces another level of complexity in the learning process and can destroy knowledge that is essential to the first task, thereby causing a redundancy in the learning processs.

f. Inability of either the user, the systems analyst (if present in the organization) or the maintenance programmer to assess the effect of the change requested on other program functions. This may later lead to costly inefficiencies as the maintenance programmer must return to the user and explore alternative changes.

2. Program Planning. The programmer gathers and surveys the program documentation that is available preparatory to planning and beginning the maintenance task. This task is approached in different ways by different programmers. Some may proceed in an orderly manner, organizing and planning the task thoroughly, while others may proceed directly to logic diagramming and coding, letting the planning occur interactively with this process. Common to any approach, however, is the necessity for the programmer to acquaint himself with the adequacy of the documentation available.

Factors in this process that inhibit programmer effectiveness are:

a. Out-of-date documentation. Changes may have occurred to the program that are not reflected in the documentation. This may lead the programmer

down false paths as he sets to the task of coding and testing changes, with an attendant loss of time and efficiency in discovering these inconsistencies and understanding and correcting them. Documentation may be out of date on several levels. First the narrative descriptions of the program logic may not truly reflect the latest status of the program. This problem may be of slight consequence if the remaining documentation is current. Second, the logic diagrams may be out of date. This can have serious consequences if the programmer makes changes based on fallacious assumptions made from such diagrams, or if he discovers inconsistencies between the diagrams and other documentation and must pause to reconcile them. Third, the source code may be out of date. This may be caused by programmers making changes directly to the object code without recompiling or correcting the source code. This will lead to programmer inefficiency if he uses the source deck to make corrections to and discovers, upon testing, that the object code derived therefrom performs in an unexpected manner. On the other hand, this could lead to a bad object code becoming the production status code, since the programmer most likely would design a test that would only verify his changes. Rectification in a production environment could be costly. Fourth, the input/output and report format documentation may be out of date, causing lost time and inefficiency while the programmer reconciles this documentation with the actual functioning of the program.

b. Missing or incomplete documentation. This problem may occur at any of the levels mentioned in the preceding section and will cause programming delay. If the programmer detects that documentation is missing, and sets about to correct this before he begins coding changes, then the consequences will be less serious than if the omission is not detected and he makes changes based on fallacious assumptions.

c. Non-standard or non-conventional documentation. If the documentation has been prepared according to standards or conventions unfamiliar to the maintenance programmer, his efficiency will be

inhibited as he attempts to translate the documentation into a form he can work with or tries to learn the unfamiliar conventions.

Non-standard (to the programmer) documentation may occur at all the preceding named levels and may entail such things as:

i. Narratives of program logic. Indicative information in unfamiliar places or format; unfamiliar computer or application terminology; unnecessary or misleading phraseology; unfamiliar or clumsy prose form; clauses that may be interpreted ambiguously.

ii. Logic diagrams. More or less detailed than the programmer is accustomed to; unfamiliar uses of diagram symbols; standard logic (such as opening and closing loops, incrementing counters and registers and input/output formatting) in unfamiliar places in the diagram or using unfamiliar symbols or variable designators.

iii. Source code. Absence of comments or, if present, using terminology not consistent with the code or unfamiliar to the programmer. Overly complex, intricate or arcane coding conventions (such as, using instruction operands for indirect addresses, blind branches caused by multiple level patches, conditional branches dependent on constant quantities, etc.). Conventional routines coded in non-standard ways, or appearing in sections of the program not expected by the programmer.

d. Absence of, or non-conventional sample runs. The programmer may look to sample runs as a source of explanation for what variables in the input will affect the output and to discern the expected performance of the program in a production environment. Absence of such documentation may inhibit his efficiency by causing him, in effect, to create such runs with some of his initial tests. Non-conventional samples may entail such factors as:

i. Runs that exhibit too small a portion of the functioning of the program and thereby don't

adequately illustrate the full range of the program functions as input and output interact.

ii. Unfamiliar terminology or symbolic conventions in accompanying documentation.

3. Test Design. After the programmer has familiarized himself with the documentation and planned his approach to the task, he then prepares a test environment for his debugging phase. This entails the extraction from the production program of a portion of the logic that he feels will adequately represent the affected program areas and the preparation of test input data. Both test programs and data may be prepared on several levels of complexity in order to simplify the programming task. At some installations test data and programs are maintained as a standard procedure, and are available at all times to maintenance programmers. Once again, the procedure in setting up test environments varies from installation to installation and with different programmers. This procedure also varies with the complexity of the change contemplated, some installations allowing changes "on the fly" to production programs if the changes are deemed to be simple enough. This process may produce factors that inhibit maintenance programmer efficiency in the following ways:

a. Machine readable input data is not available. Thus the programmer must either hand encode test data for transcription to the proper input medium, or he must write a program to generate the data. In either case his efficiency is inhibited by having to test another program or check the validity of hand encoded data.

b. Because of coding restrictions, too large a sample of the production program must be extracted to provide a proper test environment. This problem manifests itself either because the logically nested nature of the code obviates efficient testing, or because data dependencies are entwined throughout major portions of the code, or the programming language is at a level too high and too intricate to allow ready partitioning. Large sample programs may cause the maintenance programmer to look at redundant or extraneous results of test runs, and will

cause extra clerical and housekeeping functions connected with coding and testing.

c. Improper or non-conventional (to the programmer) run instructions (documentation) exist for the necessary generation of the test program object code. This may cause the programmer to generate new run instructions or to spend time in deciphering the existing ones.

4. Diagramming and Coding. The programmer prepares the initial logic diagrams and coding sheets. (It should be remarked that this and the preceding step are often done in reverse sequence, which allows the programmer to overlap the keypunching, if required, of his code with the preparation of a test environment.) This step is iterative, and the programmer during the course of a task may return to it many times to alter diagrams and coding logic. The factors that may inhibit programmer efficiency during this phase are:

a. Unfamiliarity with the language being employed in the production program. This will cause the programmer to, in effect, gain on-the-job expertise in the language being employed with the attendant loss of efficiency.

b. Unfamiliarity with the computer control procedures and language. This will create inefficiencies similar to those cited in A above.

c. Non-standard, inadequate or unconventional coding and documentation standards that the programmer must conform to. The programmer has two choices in these circumstances; he may either conform to the standards with the attendant loss in efficiency caused by his having to refer back to documentation he doesn't fully understand as he iterates through the diagram and coding phase; or, he may elect to follow nonstandard procedures more familiar to himself and later re-do them to conform to the prevailing standards. (Or he may elect not to re-do them, and thereby create the possibility that a programmer attempting to perform maintenance on his program at a later time will have his efficiency inhibited.)

d. Lack of adequate supervision or consultation resources. When the programmer is desk checking his logic, inadequate access to more knowledgeable programming staff may cause him to spend a considerable amount of time in the debugging phases that might have been eliminated by relatively short consultation.

5. Debugging. The programmer begins the testing cycle. During this phase he may, from time to time, iterate through all of the previous phases in order to arrive at an adequately debugged program.

The factors that may inhibit programmer efficiency during this phase are:

a. Improper or inadequate knowledge of program audit procedures. These procedures, in effect, organize the test phase into orderly, logical sequences of events. Lack of the use of these procedures can cause the programmer to waste time in attempting to sort out all of the logical and physical factors influencing a particular test run in order to determine the causes of bugs.

b. Lack of adequate machine generated displays at the improper termination of a test run. Dumps; input/output tape, disc and memory displays; and listings of parameter cards, are essential evidence in the detection of bugs. If these are not provided, then re-runs must be made or debugging deductions made with improper or misleading evidence.

c. Inadequate operations procedures or laxity on the part of operations personnel in following run instructions. This can cause the programmer lost time in correcting operations mistakes, in effect introducing another level of debugging; that of correcting or rectifying poor operating practices.

d. Discovery of program or data interdependencies not previously known to the programmer, that cause unanticipated aberrations in the execution of the test program. This condition can cause the maintenance programmer a loss in efficiency by requiring him to retrace his planning and orientation steps and to revise his approach to the task.

e. Unavailability of a representative sample of input data or parametric variables that exercise the program logic paths. If this inadequacy is known to the programmer, it can cause him lost time and inefficiency in hand coding data. If not known to him, it can seriously reduce the adequacy of the performance of the maintenance task, thereby creating the possibility of specious coding being introduced into the production system.

f. Inability of the programmer to test the program in a production environment. This condition is often the case for complex programming systems, particularly those that operate in an on-line or real time environment. Thus, certain conditions that cause unique paths through the system to be executed will never be encountered until the program is placed in production status.

6. Quality Assurance. The testing cycles are completed and the programmer has satisfied himself that the program is as operational as he can make it. At this point he presents the results to the customer either indirectly through an administrative chain of quality control, systems analyst or supervisory functions, or directly.

The factors that may inhibit programmer efficiency during this phase are:

a. The customer either disagrees with the results or based upon the new information obtained by reviewing the results, requests additional changes. This introduces further requirements upon the programmer for communication relating to the programming task, usually in an informal and unstructured manner, and, if the customer request is honored, may cause him to re-do significant portions of the task that he has just accomplished.

b. Lack of formal or standardized procedures for reviewing results and resolving disagreements as to adequacy of the results. This may cause the programmer to spend considerable time in arguing for, and explaining the results he has produced.

c. Detection of inadequacies in the programming results or deficiencies in the documentation accompanying them. This may cause the programmer to re-write significant portions of the program or to re-document parts of it, and could lead to an effort as significant as the original task.

7. Production Shakedown. Finally, the program is placed in production status. At this point the programmer's responsibility probably doesn't end. With the exception of those installations that maintain a group of specialist programmers for the purpose of remedying production status malfunctions, the maintenance programmer continues to have either an informal or formal continuing responsibility for the production program insofar as the changes he has made may be suspected to be causing aberrations in production runs.

Factors that may inhibit programmer efficiency during this phase are:

a. Lack of proper diagnostic techniques that allow for the detection of causes of production program malfunctions. This may cause the maintenance programmer to have to enter into debates as to the source of problems or to spend considerable effort in tracing such causes only to discover that they are the responsibility of someone else, and subject to more expedient correction by others.

b. Lack of proper definition of responsibility for production malfunctions. This will cause inefficiencies similar to those described in the preceding paragraph.

c. Lack of proper or adequate communication concerning the nature of the failure encountered. This will cause the programmer to spend unnecessary time in regenerating the conditions that caused the failure.

B. Physical Environmental Factors.

In this section are presented environmental and work factors that inhibit maintenance programmer effectiveness that apply to more than one of the activity phases described in the preceding section.

1. Clerical Functions. In the course of the maintenance programming task, the programmer is called upon to perform functions that are menial in relationship to his training and experience and that require simple skills and attention to routine that are normally the attributes required of clerks and secretaries. Factors in these processes that inhibit programmer effectiveness are:

   a. The necessity to keep track of, in an orderly manner, the many documents, source code sheet and other paraphernalia essential to the task. This requires the programmer to spend much of his time in filing, cross referencing material, and in general arranging material for ready and efficient access.

   b. The necessity to document, keep track of and otherwise arrange for easy reference, the names of variables in programs, the format and names of data elements and the sequence of source code statement numbers, etc.

2. Machine Time Availability. The lack of adequate machine time during the debugging phase may cause the programmer to lose efficiency through a loss of knowledge, between test shots, of the logical context of the phase of the test he is in.

3. Work Surroundings. The factors associated with this that can cause inhibition of efficiency are:

   a. Excess noise that cause distractions. This is particularly detrimental during periods of activity requiring intense concentration such as logic design and flow charting, coding, desk checking, data checking and debugging.

   b. Lack of adequate work surfaces and storage areas. During the course of the programming activities, many documents, manuals, and writing tools and forms must be kept track of. Lack of appropriate and adequate space to accommodate such paraphernalia cause an attendant loss in efficiency.

   c. Lack of a comfortable work area. Discomfort can cause distraction that may seriously inhibit effectiveness.

4. Inadequate Supervision and Audit. Lack of adequate supervision or audit of the programmer's progress during the accomplishment of the maintenance task. This may cause the programmer to lose time and effectiveness through lack of proper work organization or consultation on technical and administrative roadblocks.

C. Systems/Hardware Environment.

In this section various systems/hardware configurations are described and the factors that inhibit the effectiveness of maintenance programmers who have to deal with these configurations.

1. Batch Systems. The factors that inhibit programmer effectiveness in this environment are:

   a. Schedules for time on the machines have a tendency to inflexible. Since the progress of maintenance changes, particularly during debugging, is hard to predict, the programmer many times finds it impossible to get test shots at the times he needs them since production work has taken up all the available resources.

   b. Interaction with the computer is through manual methods. The programmer must fill out run instructions and submit them to operating personnel who then are expected to follow them. This introduces the possibility of human error and lost time on the programmer's part in rectifying such errors. In addition, a batch system requires more handling of cards and tapes with the possibility of error.

2. Remote Batch Systems. When the remote batch system is operating in stand-alone mode, the factors inhibiting programmer effectiveness are little different than those of a normal batch system. However, when it is interacting with a central computer for the interchange of data or programs, it introduces a new level of complexity to the maintenance programming task. The programmer must then, in addition to knowing and keeping track of the program that is the object of his task, also address himself to the interface programs. It may also be difficult, if not impossible, for him to replicate the production environment because of interference with production activities.

3. On-Line Systems. The major obstacles confronting the maintenance programmer in on-line systems is the complexity introduced by operating systems routines and the inability to fully replicate the production status environment. If real time applications are being worked on, the programmer also will have difficulty in creating test data that fully represents all of the interactive processes that may occur in production status.

II. Examples of Factors Inhibiting Maintenance Programmers as Experienced at the Chrysler Corporation.

A. Poor Communications.

1. Verbal. Although many procedures and standards existed for formal written communication, many of the vital instructions were verbally communicated. This communication medium was weakest and most detrimental in the requests for maintenance that came from the user. These requests were transmitted to an organizational group called 'Customer Service' who were, in fact, the most experienced group of maintenance programmers. Difficulties arose from the discrepancy in understanding and objectives between the user and Customer Service. The user tended to think in terms of the application with little concern for the implications to programming effort. The programmers tended to consider only the programming implications. This led to exhaustive negotiations and meetings in order to achieve agreement on the changes to be made and understanding of the resources required to make them.

2. Documentation. The documentation at Chrysler was generally up to date and conformed to well-documented standards. This was achieved at considerable cost involving training in standards, a large, well-staffed library function, and much supervisory time and effort expended in auditing documentation. The weakest point, once again, was with the user request for maintenance. The users were not trained in the documentation languages and standards, and requests had to be translated from the application language of the user to the technical language familiar to the programmer. This was the job of the systems analyst. Difficulties arose because the systems analysts had been drawn either from user organizations or from the programming staff. In either case the bias of their background tended to be translated in the work they did. Formal documentation standards were followed, but it was difficult to derive a language that was precise and inclusive of all possible circumstances. Therefore, narrative descriptions of the changes became an important part of the documentation. The ability to write

concise, unambiguous narrative descriptions varied widely among Systems Analysts. This ability did not seem to correlate significantly with technical or applications aptitude. Courses in technical writing were given periodically to Systems Analysts, but the results were not equal to the resources invested.

3. Production Bugs. The most frequent breakdown in communication occurred between the operations staff encountering bugs in production programs and the maintenance programmer responsible for correcting such bugs. Often the operator was not adequately instructed on what recovery procedures to use, and what diagnostic information to obtain. This was largely corrected by better operating procedures that instructed the operator on general procedures to follow when encountering a malfunction, better run instructions in documentation packages that covered restart and diagnostic procedures peculiar to the particular system being documented and operations turn-over training sessions conducted by the maintenance programmer for the operations staff when he placed a change into production status.

B. Inadequate Knowledge.

1. Application. The most serious effect was in communication between the user and the programming staff, as described above. Classes, conducted by members of the user's organization, were given to maintenance programmers on application topics. The main effect seemed to be an increase in the morale of the maintenance programmer and a facilitation of relations between user and programmer.

2. Program Structure. Because of the good condition of the documentation this problem was not as severe as others. It occurred most frequently with new or inexperienced programmers. It was alleviated by a formal program of cross-training on different applications, and by assigning senior 'advisors' to the inexperienced programmers.

3. Inter-Systems Effects. This was a major problem at Chrysler due to the large size and interactive nature of the applications. The systems followed the chronological flow of sales, acknowledgment, scheduling and manufacture that encompassed the

business of the Corporation. The problem occurred most frequently in emergency 'fixes' or in short duration changes. The larger changes were usually better planned and tested, since the resources brought to bear were greater. Partial alleviation of the problem was achieved by keeping a log of changes and their effects. This log was organized by major system and was kept by the programmers who were making maintenance changes to the systems. Another procedure adopted that contributed to the alleviation of this problem was systems test runs by the systems programming staff. These runs were done in order to determine data and logic inter-relationships among various systems and the results were documented and placed in the library.

4. Operations Practices. This problem manifested itself most severely in the production shakedown phase of the maintenance programmer's activities. Corrective procedures, noted above, were costly. The cost of these procedures were probably equal to the direct cost incurred without them, but the reduction in disruption of Company operations more than justified their use.

5. Programming Languages. This problem occurred most frequently with inexperienced programmers and with experienced programmers who were not familiar with the particular machine for which they were programming. It caused mistakes in coding that were often not detected until the debugging phase, and at that juncture might cause significant reprogramming. Cross training on different computers, formal programming classes, and the availability of experienced consultation alleviated these problems to a large extent.

6. Audit Practices. Unfamiliarity with these practices caused problems most frequently in the Quality Assurance phase. A programmer may have performed his task in an effective manner from the standpoint of coding and testing, but unless he could communicate, in the manner specified by standards, the results of his task to the audit team his work was not accepted. This caused some work, particularly documentation, to be redone. The problem was alleviated by including audit practices in the standard training courses.

C. Inadequate Organization and Procedures.

1. Consultation and Supervision. This problem caused the most serious consequences in the debugging phase when the programmer was attempting to trace a malfunction in his test program. Often, a few minutes with a senior programmer could solve a problem that might take hours for the maintenance programmer to solve on his own. The program planning phase could also be far less effective without adequate supervision and consultation. This manifested itself in the later phases when poor planning caused lost time while the programmer attempted to reorganize his tasks to be appropriate to the resources available. This problem was largely alleviated by adequate supervisory control (there was an average of one working supervisor for every five maintenance programmers) and the assignment of consulting duties to members of the organization that had specialties appropriate to the programming tasks. The latter procedure created problems in that it had a tendency to disrupt the work activities of the programmers who were carrying the double duty of 'consultant'.

2. Audit. Inadequate audit procedures most frequently caused problems in the debugging phase of the maintenance programmers task. This occurred because it was extremely difficult to obtain an accurate assessment of the programmer's progress during debug. The programmer typically thought that each test shot he submitted would be the last and as a consequence consistently under estimated the effort and time necessary to complete the job. In the other phases of the programming task it was fairly easy to audit programmer progress, and a very thorough audit procedure was derived and implemented.

3. Operations. This problem manifested itself most frequently in the debugging phase. It was often difficult for the programmer to acquire the test time he needed for orderly progress on the job. It was also difficult for him to communicate his specific needs to the operations organization so that the test results were optimally effective. Two organizational schemes, tried at various times, accounted for these difficulties: one allowed the programmer free access to most of the operations staff, on a semi-open shop basis, another formalized

the programmer-operations interface rigidly on a closed-shop basis. In the former case control an orderly scheduling became impossible. Many schedules were created on the basis of the aggressiveness or popularity of individual programmers, rather than on need as determined by work priority. In the latter organizational mode bureaucracy impeded progress by requiring extra effort on the programmer's part to conform to rules, and by stifling informal communication required for responsive results. The best solution that was implemented at Chrysler was something in between the two extremes: A closed shop with formal procedures, but with escape clause provided by an expediter who had informal access to all the operations staff and was the programmer's 'friend in court'.

4. Design Changes. Although design changes could require the programmer to retrace all of his programming steps, the problem most frequently manifested itself in the Quality Assurance phase. These changes were requested by the quality assurance staff because of results that didn't meet standards. No direct remedial actions alleviated this problem. The indirect actions of better training for programmers, more concise specifications at the initial request phase, and better supervision, all contributed indirectly to improving the situation.

5. Results Review. This problem was manifested primarily in the Quality Assurance phase. It was caused by an inadequate understanding among the Q/A staff, the programmer, and the user as to what should be considered as adequate results. Once again, the indirect actions cited above contributed to improvement.

6. Production Diagnostics. This problem arose during production shakedown. It was caused by inadequate production diagnostic aids and procedures for determination of causes of malfunctions. It caused the programmer and operations staff to spend unnecessary time and resources in recreating malfunctions in order to produce the proper diagnostic material. It was largely solved by better operator training, improved program documentation that

included instructions for diagnostic procedures, and better operations supervision.

7. Production Responsibility. This problem caused much confusion and lost time when a production program malfunctioned. No one wanted to claim responsibility for correcting the malfunction and much energy was expended in fixing responsibility. This was greatly improved by adopting the following practices: at the conclusion of a maintenance programming task, the Quality Assurance function assigned an integer to the program. This represented 'n' production cycles through which the program was run before the maintenance programmer was relieved of primary responsibility for any malfunctions. After that, the responsibility resided with the systems programming staff.

8. Clerical Duties. This problem was most inhibiting during Diagramming and Coding, and Debugging phases. It arose because the maintenance programmers had so much data and documents to keep track of and to organize for effective work activity. Since Chrysler's documentation was extensive, and the systems were massive and complex the amount of paper that a programmer used in accomplishing a maintenance task was considerable. Further complicating the clerical task was the necessity to conform to detailed and extensive standards. This required the programmer to organize his work so that its completion would yield results consistent with the standards. In the other phases of the programming task this problem was largely alleviated by the use of 'para-professionals' to assist the programmer in the clerical functions. These personnel were secretary/clerks and programmer-trainees. For each group of 5 programmers there was at least one such person assigned. Approximately half of their time was devoted to organizing material that the programmer required to accomplish his task, and preparing data and keeping track of material as it flowed from the programmer to the different organizations. However, in the coding and debugging phases the programmer's use of such materials was so interactive and random that it was difficult if not impossible to utilize such help. The only step that was taken to improve this was the incorporation of clerical methodology in the standard training courses.

D. Missing Aids.

1. Production Environment Test. Since the systems at Chrysler were large, complex and highly interactive, it was often impossible for the maintenance programmer to have all of the production environment conditions present for testing. This led to much extra effort in attempting to design tests that were as close as possible to the production environment, and caused bugs to be left in programs that weren't detected until the program was in production status. This problem was never satisfactorily solved but certain measures were taken that improved the situation. The systems programming staff extracted from production runs statistically representative samples of input data and created test files from these. They also created simulated interaction programs that allowed the programmer to test interactions without the full system. Obviously, such test files and simulations could not be exhaustive, and the problems continued, particular in production programs.

2. Machine Displays. The largest problem at Chrysler was the unavailability of the correct selective dumps of core or external storage media in order to pin-point the cause of program malfunctions. Frequently, when using selective dumps, the programmer would get the wrong area of storage, or he would get too large a dump and have to waste considerable time in finding the area he was most interested in. Another problem was presented by the fact that many times the dump was taken when storage was in a different state than at the time the malfunction occurred. Trace routines were used extensively in an attempt to reduce the effect of this problem, but the routines introduced other problems. They slowed down execution, created unwanted displays, and were tedious to set up. No conclusive solution was ever found for this problem.

3. Test Data. Even though systems test files existed for all major systems, they were not all inclusive and the programmer was often faced with the task of generating his own test data. This occurred often enough that the expense involved amounted to approximately 10% of the average programmer effort expended on maintenance tasks. Creation of automatic

and selective data extraction and formatting routines alleviated this problem slightly but the effect on effort was negligible since the routines required effort to understand, use and maintain.

E. Inadequate Environment.

1. Machine Time. There were two conditions that occurred at Chrysler that were detrimental to programming effectiveness; first, when there wasn't adequate machine time to allow the programmer to get tests back fast enough so that his time was fully occupied during debugging, and second, when there was almost limitless test time and the programmer had a tendency to submit tests without adequately desk checking the previous test. The latter situation caused the programmer to become confused and disorganized in his testing approach. The attempt at solution caused a study to be conducted, and periodically updated, for the most cost effective average turnaround duration. When this average fell below cost effective tolerances, and it was determined that the condition wasn't transitory, additional equipment was installed. Another procedure that alleviated this problem was the introduction of a priority system that took into account the importance of the application to the company and also incorporated an aging system that increased the programmers' priority as the interval between test shots increased.

2. Work Surroundings. This problem was most debilitating during those programming activities requiring high concentration, such as coding and debugging. It was caused by poorly organized, noisy work space. The programmers were in a 'bull pen' arrangement with tile floors, non-acoustical ceiling and no provision for meeting rooms for consultations. This problem was largely alleviated by organizing the work space into well arranged two-man cubicles, carpeting the floors and installing acoustical tile on the ceilings. For every four cubicles, there was a small six-man conference room. The reduction in distractions and disorganization improved output of the programming staff by a measured 8%.

## III. Analysis of Inhibiting Factors.

### A. Inhibiting Factor Scoring System.

The following system is arbitrary and subjective. It is designed to assign relative rankings to inhibiting factors as experienced at the Chrysler Corporation. It assumes a non-linear relationship between effects that inhibit but does not take into account the possibility of interrelated effects. (e.g., if an inhibiting factor $x_1$ exists in an activity area $y_1$, with a rank score $r_1$, then it is possible that there exists $x_2$, $y_2$ such that $r_2 = f(r_1, x_1, y_1)$. The determination of such relationships would require extensive research.)

The efficacy of the system described below is that, by assuming additive effects of rank scores, it gives a ranking of seriousness to inhibiting factors and activities as they potentially contain such factors; and it is easily understood.

| EFFECT | WORST SCORE |
|---|---|
| 1. Causing major disruptions felt widely throughout both user and programmer organizations | 6 |
| 2. Causing disruptions felt significantly beyond the programmer and his immediate associates, but not in the user organization | 4 |
| 3. Causing disruptions felt only by the programmer and his immediate associates | 2 |
| 4. Causing disruptions felt only by the programmer | 1 |

The scores reflect relatively perceived disruptions. It might be analytically more satisfying to reflect absolute cost of resources. However, this measurement is not available. The assumption, then, is that cost of disruption is proportioned to the extent of organizational reaction. This assumes a rationally responding and perceiving organization. The scores indicated are the "worst" effect that a factor might create in an

organization. In the matrix that follows, scores are multiplied by a probability coefficient which indicates a guess, based on experience, as to the probability of such a factor having effect or occurring in a given activity. (Zero probabilities and scores are not shown.) The resulting product produces a rank score. Rank scores are then summed by row and column to produce summary rank scores for activities and factors. These are listed in this Appendix and in Chart 2.4.3 of Part I of the Phase One Report.

| FACTORS \ ACTIVITIES | INITIAL REQUEST | SYSTEMS ANALYSIS | PROGRAM PLANNING | DIAGRAMMING & CODING | TEST DESIGN | DEBUGGING | QUALITY ASSURANCE | PRODUCTION SHAKEDOWN | FACTORS RANK TOTALS | RANK |
|---|---|---|---|---|---|---|---|---|---|---|
| POOR COMMUNICATIONS | 7.2 | 2.8 | 1.0 | 1.4 | 0.0 | 1.6 | 2.0 | 4.8 | 20.8 | |
| VERBAL | 6x.8=4.8 | 4x.4=1.6 | | | | | 4x.2=.8 | 6x.2=1.2 | 8.4 | 3 |
| DOCUMENTATION | 6x.4=2.4 | 4x.3=1.2 | 2x.5=1 | 2x.7=1.4 | | 2x.8=1.6 | 4x.3=1.2 | 6x.2=1.2 | 10.0 | 1 |
| PRODUCTION BUGS | | | | | | | | 6x.4=2.4 | 2.4 | 17 |
| INADEQUATE KNOWLEDGE | 1.8 | 3.2 | 0.8 | 1.8 | 0.4 | 5.8 | 5.2 | 7.8 | 30.8 | |
| APPLICATION | 6x.1=.6 | 4x.2=.8 | | | 4x.3=1.2 | 2x.5=1 | 4x.3=1.2 | 6x.2=1.2 | 6.0 | 7 |
| PROGRAM STRUCTURE | 6x.1=.6 | 4x.8=1.6 | | 2x.3=.6 | 2x.3=.6 | 2x.3=.6 | 4x.2=.8 | 6x.2=1.2 | 6.0 | 7 |
| INTER-SYSTEMS EFFECTS | 6x.1=.6 | 4x.2=.8 | | | 4x.2=.8 | 2x.3=.6 | 4x.3=1.2 | 6x.3=1.8 | 5.8 | 8 |
| OPERATIONS PRACTICES | | | | | 2x.5=1 | 4x.5=2.0 | | 6x.2=1.2 | 4.2 | 12 |
| PROG. LANGUAGES | | | | 4x.3=1.2 | | 4x.4=1.6 | | 6x.4=2.4 | 5.2 | 10 |
| AUDIT PRACTICES | | | 4x.2=.8 | | 4x.2=.8 | | 4x.5=2 | | 3.6 | 14 |
| INADEQUATE ORG./PROCDRS | 0.0 | 0.8 | 1.6 | 3.0 | 2.0 | 8.0 | 4.2 | 10.4 | 30.0 | |
| CONSULTATION/SUPV. | | 4x.2=.8 | 4x.4=1.6 | 2x.5=1 | 2x.4=.8 | 4x.5=2 | | 6x.1=.6 | 6.8 | 6 |
| AUDIT | | | | | 4x.3=1.2 | 4x.5=2 | | 6x.2=1.2 | 4.4 | 11 |
| OPERATIONS | | | | | | 4x.5=2 | | 4x.2=.8 | 2.8 | 16 |
| DESIGN CHANGES | | | | | | | 6x.2=1.2 | | 1.2 | 18 |
| RESULTS REVIEW | | | | | | | 6x.5=3 | | 3.0 | 15 |
| PROD. DIAGNOSTICS | | | | | | | | 6x.6=3.6 | 3.6 | 14 |
| PROD. RESPONSIBILITY | | | | | | | | 6x.7=4.2 | 4.2 | 12 |
| CLERICAL DUTIES | | | | 2x1=2 | | 2x1=2 | | | 4.0 | 13 |
| MISSING AIDS | 0.0 | 0.0 | 0.0 | 0.0 | 2.0 | 4.8 | 5.4 | 8.2 | 18.4 | |
| PROD. ENV. TEST | | | | | 2x.5=1 | 4x.5=2 | | 4x1=4 | 7.0 | 5 |
| MACHINE DISPLAYS | | | | | | 4x.2=.8 | 2x.2=.4 | 6x.7=4.2 | 5.4 | 9 |
| TEST DATA | | | | | 2x.5=1 | 4x.5=2 | 2x.5=1 | | 4.0 | 13 |
| INADEQUATE ENVIRONMENT | 0.0 | 1.0 | 2.0 | 2.0 | 2.5 | 3.0 | 0.0 | 6.0 | 16.5 | |
| MACHINE TIME | | | | | 1x.5=.5 | 4x.5=2 | | 6x1=6 | 8.5 | 2 |
| WORK SURROUNDINGS | | 2x.5=1 | 2x1=2 | 2x1=2 | 2x1=2 | 1x1=1 | | | 8.0 | 4 |
| ACTIVITIES RANK TOTALS | 9.0 | 7.8 | 5.4 | 8.2 | 10.9 | 23.2 | 13.8 | 37.2 | | |
| RANK | 5 | 7 | 8 | 6 | 4 | 2 | 3 | 1 | | |

B. Matrix of Activities vs. Factors.

C. Summary of Rank Scores by Rank & Percentages.

1. Inhibiting Factors.

| Rank | Factor | Score | % of Total | Cum. % |
|---|---|---|---|---|
| 1 | Documentation | 10.0 | 8.7 | 8.7 |
| 2 | Machine Time | 8.5 | 7.4 | 16.1 |
| 3 | Verbal Communication | 8.4 | 7.4 | 23.5 |
| 4 | Work Surroundings | 8.0 | 7.0 | 30.5 |
| 5 | Prod. Env. Test | 7.0 | 6.1 | 36.6 |
| 6 | Consultation/Supv. | 6.8 | 6.0 | 42.6 |
| 7 | Application Knowledge | 6.0 | 5.2 | 47.8 |
| 8 | Inter Systems Effects | 5.8 | 5.0 | 58.0 |
| 9 | Machine Displays | 5.4 | 4.7 | 62.7 |
| 10 | Prog. Languages | 5.2 | 4.5 | 67.2 |
| 11 | Audit Procedures | 4.4 | 3.8 | 71.0 |
| 12 | Operations Practices | 4.2 | 3.7 | 74.7 |
| 13 | Clerical Duties | 4.0 | 3.5 | 81.9 |
| 14 | Audit Practices | 3.6 | 3.1 | 88.5 |
| 15 | Results Review | 3.0 | 2.6 | 94.2 |
| 16 | Operations Org./Prcdrs. | 2.8 | 2.4 | 96.6 |
| 17 | Production Bugs | 2.4 | 2.2 | 98.8 |
| 18 | Design Changes | 1.2 | 1.1 | 99.9 |

2. Inhibiting Factors by Groups.

| Rank | Group | Score | % of Total | Cum. % |
|---|---|---|---|---|
| 1 | Inadequate Knowledge | 30.8 | 26.9 | 26.9 |
| 2 | Inadequate Org./Prcdrs. | 30.0 | 26.1 | 53.0 |
| 3 | Poor Communications | 20.8 | 18.4 | 71.4 |
| 4 | Inadequate Environment | 16.5 | 14.6 | 86.0 |
| 5 | Missing Aids | 16.4 | 14.2 | 100.2 |

3. Activities

| Rank | Activity | Score | % of Total | Cum. % |
|---|---|---|---|---|
| 1 | Production Shakedown | 37.2 | 32.4 | 32.4 |
| 2 | Debugging | 23.2 | 20.3 | 52.7 |
| 3 | Quality Assurance | 13.8 | 12.0 | 64.7 |
| 4 | Test Design | 10.9 | 9.4 | 74.1 |
| 5 | Initial Request | 9.0 | 7.8 | 81.9 |
| 6 | Diagramming & Coding | 8.2 | 7.1 | 89.0 |
| 7 | Systems Analysis | 7.8 | 6.7 | 95.7 |
| 8 | Program Planning | 5.4 | 4.7 | 100.4 |

# APPENDIX V

## CSA Software Maintenance Report

# SOFTWARE MAINTENANCE

## Purpose of the Report

This brief report is a summation of CSA's experience with and analysis of software maintenance problems. The report also attempts to identify the computerized tools now utilized and the potential areas for improved computerized tool development in support of the maintenance programmer. Further, recommendations are made for research areas in the development of these computerized tools. The prime focus in the report is on the maintenance programmer in a large real-time software system.

The resources utilized for this report have come from three sources:

1. The experience of the authors.
2. Material gathered and reviewed in a literature search on the software maintenance problem.
3. Interviews of maintenance programmers, primarily at SDC, in the last couple of months.

The authors, Levi J. Carey and Willis Hudson have had extensive experience in software maintenance. Both were employed at the System Development Corporation during a period of five years when software maintenance of the SAGE and BUIC systems was an awesome chore. SAGE and BUIC are both large real-time command and control systems. Their responsibility was to develop tools for software maintenance. One of the tools mentioned by an SDC programmer in the latest series of interviews was originally developed and conceived by the authors--the symbolic corrector program. The SDC programmer stated that the corrector program was the most valuable tool they had. In any case the authors have had ample opportunity to view the software maintenance problem as maintenance programmers and as tool developers.

The literature search has provided information primarily on the tools that are presently available or are being researched. Later in this document these tools are identified. Interviews with present SDC employees and with other personnel have identified further some of the major problems of software maintenance. One of the disappointments of the interview technique, however, was the lack of insight into what might be done to change things. Indeed, there appears to be some complacency or resignation in

the maintenance programmer's attitude relative to what might be done for him.

## What Is Software Maintenance?

Software maintenance is that activity required to support the use of the software system. Generally, software maintenance activity can be organized into two types. These are:

1. Software modification

2. Software repair

These are the primary activities of the maintenance programmer. However, there are other conditions which also require servicing. They include the requirement for the programmer to provide information about the system's operation, also to provide improvements in system operation. The latter two requirements are generally of an ancillary nature and are prerequisites to program modification. Requests for modifications to the system require the majority of the maintenance programmer's activity. He is usually bombarded with modification requests to accommodate either: the user, some system component (usually hardware) that is malfunctioning or could malfunction, and changes in the system environment, i.e., new requirements. The other activity required of software maintenance programmers is error correction and program repair. If the user or anyone else discovers a program error, one of the maintenance programmer's primary tasks is either to repair the software system or to devise a way around the failure which is satisfactory to the user. Error correction for large real-time programming systems consumes a considerable amount of the maintenance programmer efforts and is used to justify the considerable expense of retaining some programmers in this type of maintenance effort on a full-time basis.

These two areas, program modification and program repair, will be the primary areas of discussion in this report from the viewpoint of the maintenance programmer we are attempting to aid.

## Software Modification Process

Any chronology of the maintenance programmer software modification activity would include the following six phases:

1. Request for change;

2. System change design;

3. Detailed program change design;

4. Program code development;

5. Program testing;

6. System validation.

Request for Change

The request for change from our experience (and it also appears to be the experience of others interviewed) varies greatly in terms of: the magnitude of the request for change; the procedures used to process the change; and the knowledge of the requester relative to the software system.

Requests for changes, when formally submitted, have usually had some informal analysis. From the maintenance programmer's point of view, his first task after receiving the request for change in a large real-time system is to determine what part of the system is involved and what programmers or analysts should be involved. The programmer receiving or coordinating the request for change first attempts to identify from the statement of the change what primary data structures are involved and what computer program modules are involved. He generally has some knowledge of the areas that are involved. For instance, he is probably aware if the change involves some large functional area of the system such as displays. He then goes to the lead programmer in that area for further information. Notice that the system of discerning information from the written statement of the request for change is based largely upon human theory. This part of the modification process is subject to considerable error - catastrophic error! The programmer receiving the request for change may overlook a vital area that should be involved.

One of the most effective aids to the maintenance programmer in processing a request for change is a program which builds an alphabetized cross referencing index between the data names and a description for their reason for being. This is organized as a data base of information in the SAGE system and it was called COMDOC or COMPOOL documentation. It can be extracted from the program code. Another aid which was used is a system set/use. This program provides cross referencing between names and the program module which either sets the data or uses the data. With these types of automated documentation, the programmer receiving the request for change has a better idea of where to go to find the affected parts of the system and the affected programmer.

## System Change Design

After is has been established which programmers are involved and what part of the system is involved, a meeting of the personnel involved is usually called to decide upon a design for the system modification. Generally there were several alternatives. Optimization of several design goals is attempted at once--the foremost consideration is to implement the change and not degrade system operation; further, to implement the change in such a way that a minimum of storage is utilized and minimum operating time is realized. Another goal is to implement the change so that the work load is evenly distributed thus facilitating production of the change. Once a design has been agreed upon, one of the larger and more onerous chores is to document it.

Computerized tools used to support the system change design activity include those which were also used to support the request for change analysis, namely COMDOC and a system set/use. There is considerable reliance upon non-computerized documentation during this phase. This includes the Operational System Description Manuals, Part I Specifications and Part II Specifications where and if they exist. Again the primary tool in the design activity is the programmer's memory. If the change requires data restructuring or program module reallocation, the systems are used and are very valuable. If the change is of sufficient magnitude, a simulation of the modification may be required. These simulations are rarely of a general nature, but are customized to the system being modified.

## Program Change Design

The individual maintenance programmer usually has the responsibility of developing a design change for one or more program modules. This is accomplished usually by the programmer outlining in an informal manner the areas of the Part II documentation that are to be changed. He is generally requested to identify at this time his system test requirements. This is infrequently accomplished. The program design activity is highly dependent upon how intimate the programmer is with the program module for which he is responsible. Modification to a computer program always involves some risk due to the fact that incomplete knowledge may generate another error.

One of the computerized tools used during the program design process is cross-referencing programs which relate data names to statements where they are used and other identifiers in the

program such as statement labels to their references throughout the program. Very few flow charts generally exist in Part II Specifications. If flow charts are automatically derived they are rarely of much aid except for program structure analysis. One of the problems in performing modification to programs is to obtain a listing of the source program as it is on the operational tape. If the program has been patched with corrections then there is always a risk that the corrections have been improperly noted or not noted at all. One of the more effective tools which the authors conceived and developed for assembly language programs was a disassembler--a program which took correction patches and integrated them into a program listing so that a reliable listing of the computer program at the source language level existed. For higher order language programs, a decompiler or un-compiler is required. This was also attempted with less success. Some parts of the system were in a higher order language. We were not successful for several reasons. The most prominent reason is that it was difficult to allow a programmer to code in a higher order language and then integrate the patched code into the binary programs. This was much simpler in assembly language programs. We will say more about this later.

## Program Code Development

The maintenance programmer is required to make a decision as to how to proceed in developing code to implement a modification. He generally has two methods available to him--a program patch or reassembly/recompilation. The program patch method has the advantage of isolating the program modification and quick implementation. The reassembly or recompilation has the advantage of coding in the source language of the program and an ability to reorganize the program. If the proper tools are not available, program patching will degrade the system documentation. Also, if a program patch system is not available in the source language, the mechanics of making the change may be cumbersome, i.e., octal corrections. In any case, code must be developed either for a patch or a reassembly or recompilation. This part of the task generally requires less time than most other processes; however, it is one that is considered first when program maintenance is discussed.

The tools used for program coding are symbolic or octal corrector programs for patching and the assembler and compilers for source program translation. Cross-referencing programs are also used to aid in determining how to code the change.

Allocation programs for data are utilized when a change involves reallocation of storage. A considerable amount of program modification involves recoding variable names, dimensions, locations, etc. The use of COMPOOL or data base generation programs for data names and descriptors is very effective in facilitating modifications. In a large number of cases, all that is required is a data change.

## Program Testing

The area of program testing on a program module basis is usually left up to the individual maintenance programmer. This is generally accomplished in a system environment however; that is, the modified program is placed in the system and checked out using the system's tools. There is considerable advantage to this for the maintenance programmer. Essentially he has available to him the system's simulation and data reduction facilities. Checkout for the modification is most often accomplished by generating inputs for a program module for a particular set of functions and examining the results. On the first few checkout runs, typically the program does not run to completion. However, after that hurdle has been passed the problem quickly blossoms into one of generating test inputs and reviewing test outputs. Since a considerable amount of time is consumed in this activity and since this activity is less systematic than CSA believes it might be, particular emphasis should be placed on improved testing.

The tools utilized for program testing are data conversion programs which accept decimal numbers and associated variable names as input and convert these to binary for system use at the proper time. Tools for high level simulation are also used. A function description is input and variable values are generated. Data reduction programs of considerable variety are utilized during the test process. During the debugging process several kinds of cross-referencing and post auditing tools are used.

## System Validation

System validation is the most constraining factor in the production of a software modification. Validation is a "forcing function" requiring software modifications to be bundled permitting the system to remain stable for a period so that system validation may be applied. The maintenance programmers when faced with software system validation tasks generally organize themselves so that either a separate group is responsible for this

activity or a subset of the implementation group is responsible for this activity. It is approached most usually in a large real-time system in a formal manner in that test plans are devised to exercise the area changed. Further tests are usually maintained for general system exercise. Maintenance programmers' success using these techniques have been less than optimum as evidenced by the numerous discrepancy reports which are immediately generated after validation of a new version or model of a system. Indeed, most users would rather run with an error (if it is not extremely serious) in a system with which they are familiar than to modify the error and risk introducing new errors.

The maintenance programmer begins by generating simulation inputs for the software system. The level of the simulation input language is important to the maintenance programmer because it defines the amount of work he must supply in generating a system exercise. Simulation inputs which require variable names and data values are most difficult to generate but are also useful for precisely controlling the input. After generating simulation input and exercising the computer program, the next task is data reduction. It has been the author's experience that data reduction in terms of the number of statements gets to be the biggest part of almost every software system. The reason is the requirement for variety and type of output. If the maintenance programmer must use a memory dump, he is indeed in a sorry condition. Generally, however, large real-time systems have some data reduction facility; very rarely at a function level however. Data reduction is provided for variable names and values and it is also provided for some functional areas that are easy to decode such as keyboards, switchboards and some displays. Data reduction for air defense systems on hostile and interceptor conditions is difficult to come by.

Computerized tools which support system validation generally are of the type to aid the mechanics of program testing. Very rarely are tools available which attempt to assess program correctness.

One tool is generally available; it is an audit type program. This program indicates to the user which statements or subprogram modules have and have not been exercised.

## Tool Utilization

Following is a list of tools which are presently used and identified for the phase of software maintenance activity they support.

| | |
|---|---|
| Request for Change | Cross-referencing Index<br>Data Dictionaries<br>System Set/Use |
| System Change Design | Data Name Description<br>System Set/Use<br>Data and Module Allocation Systems<br>Analytic Simulation |
| Program Change Design | Data Name Statement Cross-Referencing<br>Data Dictionary<br>Program Set/Use<br>Automatic Allocators<br>Flow-chart Generator Program |
| Program Code Development | Assemblers, Compilers<br>Dis-assembler/Uncompiler<br>Symbolic Corrector Program<br>Machine Language Corrector Programs<br>Data Base Audit Programs<br>Automatic Update Programs<br>Code Analysis and Improvement Programs<br>Reformatting Programs |
| Program Testing | Simulators, Functional and Data Oriented<br>Data Reduction<br>Trace and Trap Routines<br>Timing Analysis Programs<br>Data Analysis Programs<br>Initialization Programs<br>Standard Test Libraries<br>Test Drivers<br>Data Generators |
| System Validation | Register Use Analysis<br>Statement Use Analysis<br>Data Reduction<br>Standard Test Library<br>Regeneration of Functional Statement |

## Potential Areas for Computerized Tool Improvement

The authors see six areas in which computerized tool development will aid the maintenance programmer significantly. These are, in their order of importance:

1. Techniques for computerized documentation support.
2. Techniques to support program testing and validation.
3. Techniques to support program debugging.
4. Techniques for coding of modifications.
5. Techniques for improved and generalized simulation and data reduction.
6. Techniques for automated system allocation.

## Computerized Documentation Support

Two different types of documentation requirements exist for the maintenance programmer. The first and the one with which he is most intimate is the documentation directly related to the program code, i.e., cross-referencing listings, program listings, data name listings, etc. The second type of documentation relates to system requirements, i.e., operational handbooks, Part I Specifications, Part II Specifications, etc. Techniques generally are available for the generation of most of the documentation of the first category; however the documentation is degraded after the system has been modified. Further, considerable improvement in this documentation for the individual program module could be made. The authors believe that the second type of documentation, requirements and design documentation, can be made much more usable through computerized techniques. What is required is a thread of continuity through the computer program, the Part I and Part II Specifications, and the operational system description. All of this documentation however can benefit from the use of interactive consoles and a common data base.

## Computerized Support For Testing and Validation

Techniques which support program testing and validation have recently taken two paths: 1) development of formal program proofs and associated translation programs and 2) measures of

program testing required and applied for software reliability using computer aided tools. The authors feel that what is needed now is further development of the latter technique. Quantitative data on the correlation of software reliability and validation is required. Real progress is software validation and verification or software reliability can only be made when it is known what is significant about software reliability; numbers assigned to the effects and progress measured using these numbers.

## Computerized Support for Program Debugging

This is an area in which an investigation and projection of techniques likely to be used in the future should be pursued. Coding techniques and the support tools differ vastly depending upon the approach one takes for software modification. There are two techniques: 1) patch the computer program and 2) decompile and reassemble the program. Patching a program requires a completely different set of approaches and tools--some quite sophisticated. In spite of claims to the contrary, patching of computer programs has remained as a primary technique for program modification. Recompilation or reassembly of programs has disadvantages in the present state of the art relative to software reliability and computer time utilization. One other consideration should be made for coding of modifications. This is a projection of the level of languages likely to be used in the future. Decisions relative to whether our attention should be focused on assembly, procedure, or problem oriented languages, or all three, are analyzed.

## Improved Simulation and Data Reduction

Techniques for establishing generalized simulation and data reduction facilities to facilitate program testing should be investigated. An identification should be developed of the levels of language input and definition of the types of output more than likely to be required.

## Automated System Allocation

Although not one of the areas which is more prominent, a software system modification involving reallocation or resegmentation of data and program modules is usually a very cumbersome and long lead time chore. Recommended here is the investigation of what

techniques are presently in existence, their effectiveness, and the potential for generalizing these capabilities.

## Conclusion

The six areas identified for potential improvement using computerized tools not only supply the maintenance programmer with more powerful system software, but will probably change his responsibilities and methods of operation. Further, tools of the type recommended benefit the production programmer equally well.

# APPENDIX VI

- Scenarios and Questions
- Higher Order Languages and Maintainability

CONTENTS

## Part I

## Introduction

The following scenarios were developed by two project staff members. Their mission was to develop descriptions of "ideal" maintenance programmer environments that were as free of efficiency inhibiting factors as their imaginations would permit. From these scenarios was then developed a list of questions that would have to be answered to determine the feasibility and efficacy (in removing fundamental factors) of the ideal environments.

In creating the scenarios the staff members used the questionnaire (Appendix A), The Chrysler Study (Appendix C), The Buic Case Study (Appendix B) and their own experience as background information.

The "position" paper (Part VI of this Appendix) was developed by a staff member in order to examine questions relevant to higher-order languages and their maintainability. It is in a different format than the scenarios because it was felt that, for this topic, an expository approach would be more realistic because of the infeasibility of developing specifications for a new higher-order language.

## Part II

## Flowcharting Scenario

### Introduction

At the highest level, flowcharts provide insight into relations of major system components. In this scenario these are available on-line with boxes keyed by pointers to high-level, code-related documentation. If there is more than one level of flowchart, the boxes of the $n^{th}$ level expand into the $n + 1$ st level flowcharts and the programmer can move back and forth on the console between these levels by the implicit use of expansion and keyword oriented pointers.

### Programmer Scenario

The maintenance programmer (MP) works with a console CRT display unit with light pen attachment. By depressing certain keys on the console he has displayed on the CRT the "table of contents" of the documentation system. The table first displayed lists all of the "major" system titles of documentation

contained in the system, (e.g., "Payroll", "Accounts Receivable", "LP Model of Plant Production", etc.) Beside each title are symbols that denote the status of the system name. The symbols denote status such as "documentation in the process of being brought up to date", "major (minor) revisions in process", "system current", "system locked up-see system authority for permission to access", etc. At this point the MP has his choice of two possible actions; first, he may elect, through a touch of the light pen to the appropriate systems title, to have displayed a more detailed or expanded "table of contents" for that particular system or, second, he may elect, through a touch of the light pen in a different place on the appropriate systems title, to have the top level flowchart for the named system displayed (assuming it's not "locked-up"). In the first case, there is displayed on the CRT an expansion of the "table of contents" of the system named, which lists all of the subsystems of the major system and denotes, with the symbols described above, the system's status. In the second case, a top-level flowchart is called forth on the CRT and displays the gross systems interrelationships in standard flowcharting notation.

Each of these options present two options of their own. In the case of the display of the more detailed "table of contents" the MP can elect, by touching the light pen to the appropriate sub-system title, to have the top level flowchart for that sub-system displayed on the CRT. In the case of the "gross" overall system flowchart he may elect, by touching the light pen to the appropriate flowchart box, to have that box "expanded". Expansion displays on the CRT the next level of detailed flowchart that describes the top level box in more logical detail.

At any level of display on the CRT, except the topmost level, the MP may elect to "contract" instead of expand. This is done by depressing a "contract" button on the console and the next higher level of documentation from where the presently displayed documentation emanated, is displayed. In the case of flowcharts, this action would display the next higher level that would, in general, contain the presently displayed documentation represented as one figure. In the case of "tables of contents" the contract action would display a table that would contain the presently displayed table represented as one line.

Through the exercise of the appropriate options the MP may progress through higher or lower levels of detail in flowcharting documentation and relevant tables of content. In addition the MP has, at any level except the topmost, two more

important capabilities. The first is called "Save". Through the depression of buttons on the console the level that is presently being displayed may be saved in the sense that at any later time the MP can depress another button on the console and have displayed the documentation that was saved without having to return to it a level at a time as described above. Several (perhaps as many as 10) saves are available. A complementary capability available to the MP is that of displaying at any time without interrupting the sequence of his access to documentation, the contents of all saves. Upon depression of the proper button on the console a table of saves is displayed that contains the save number, the title and level of the documentation saved, and a "remarks" section which contains space for a dozen or so characters that the MP may input at the time of the save.

The second capability is that of "scan". At any time that the CRT is displaying flowcharting documentation, the programmer may elect to "scan" by depressing a scan key on the console and pointing the light pen at one of the borders of the screen to indicate the "scan" direction. The CRT will, in effect, then act as a "window" on a "universe" represented by the total documentation available at the level presently being accessed. By pointing the light pen at different screen borders the MP will cause the documentation to "shift" in that "direction". Before he starts this operation an automatic save is done by the system so that he can always return to the window "scene" he started with.

Systems Scenario

The flowcharting system is designed around a central executive or handler routing that responds to inquiries on the CRT and consoles. It also handles message interpretation, formatting for displays, pointer and register maintenance and other housekeeping chores.

The systems files are constructed in four major files: the flowchart file, containing the codes that can be translated by the handler into figures and the text that appears in the diagrams; the pointer file that gives the addresses and logical linkages for the flowcharts and also contains the tables of contents; the scan file which contains pointers that indicate, directionally, the linkages for each figure in each level of documentation; and the save and systems dynamic files which contain the pointers for saves that the MP has made and also contains registers and variant systems addresses for systems returns to routings in process, interrupt status

switches, etc.

Upon the initial request by the MP for the top level table of contents the handler sets up a work space identified with the console initiating the request. It clears the work space or saves it in archival storage depending on an option exercised by the MP at sign off time. While interacting with the MP at the console the handler keeps track of "where it is", "where it's been" and "where it's going". "Where it is" is kept track of by loading a systems register with the address of the pointer set that describes the locations of the elements that make up the documentation currently being displayed. "Where it's been" is available to the handler by accessing the above mentioned pointer, going to the pointer file and reading the referal pointers for the next higher level of documentation in the file. "Where it's going" is available by going to the file in the same manner and reading the referal pointers for the next lower level of documentation.

Questions raised by the flowcharting scenario.

FC1 Can the MP work without hard copy? Can he use only the CRT displays when examining and working with flowcharts? If he requires hard-copy, when and to what extent will he require it? When does the amount of hard copy he might generate exceed the usefulness of the automated system, in particular, the "save" feature?

FC2 What effect will out-of-date documentation in the file have on the MP's effectiveness? Statistically, how "out-of-date" might a file be at any one time compared to manual hard copy files? What is the benefit that can be ascribed to the difference, if any?

FC3 Is it possible to define a "unit" of flowcharting logic that will allow the MP to access displays that are defined for him and the system consistently? Should a unit defining algorithm be subjective (i.e., dependent on the particular system) or objective (i.e., independent of the system and thereby requiring some degree of conformity to a standard procedure for flowcharting)?

FC4 What hardware characteristics facilitate the flowcharting system? Are these characteristics available in existing hardware? If not, what are the specifications for new hardward designs and what are the cost/benefit trade-offs? (Some examples of facilitating hardware characteristics: high speed buffers for address "saving", heirachial file organization with hard-

wired dynamic storage allocation, etc.)

## Part III

## Structured Documentation Scenario

### Introduction

Structured Documentation (SD) assumes a programming language environment with the characteristics of the structured programming sort--the documentation occurs in a hierarchal structure keyed to major down through minor "documentation units" which are both syntactically and semantically defined. This documentation would probably be kept in a file separate from the source language text to which it pertains so that it becomes a more tractable data base for meeting other needs. These needs are described below. The system of pointers may itself be separately filed.

a) "Expando Text": Keywords or phrases in the blocks of documentation may be light-penned to cause the indicated entity to expand into more detailed sentences, paragraphs and/or tabulations. The fully expanded explanation may represent the original, programmer-submitted text which has been boiled down in several successive stages by (an) automatic abstracting algorithm(s).

b) Programmed Learning: A system to provide a self-taught beginner's or refresher course in gross or fine system details can exploit the SD data base by making use of a pointer structure keyed to concepts, keywords, system functions and data names.

c) Inquire: A similar pointer structure or a subset of the pointer structure of b) can be used to provide answers to specific inquires, e.g.,

- all uses of a variable or data name
- all I/O statements referring to a device or datum
- the declared type of a variable, etc.

d) Management Information: The pointer connection of the documentation tree to the source code tree can be exploited to alert both the programmer and management to which parts of the documentation require updating.

Management can check that this has, in fact, been done and done well.

e) Selective Dissemination of Information: The structure mentioned in d) can be combined with a current table of "who's working on what" (which might be generated from sign-in name data) to provide notices to those (other than maintenance programmers) most likely to be affected by changes to the program.

f) Historical Data: Maintenance programmers express a need to know not only the what, how and why of a program but also some perspective on the history of changes and modifications it has undergone (and who did them). This historical data can be generated as changes are made and approved and stored on relatively slow, cheap archival storage devices. This data can be fetched to re-create the portion of code and documentation as it was at each stage of its evolution.

Work on a system having some of these capabilities was done by Nathaniel Rochester of IBM, Cambridge, Mass. IBM regards this work as experimental and proprietary and refuses to divulge any information on it.

UNIVAC'S EXEC 8 operating system permits one to store up to 1000 program modification "cycles" and to generate therefrom any intermediate version of the program. Although this feature worked, it was never observed to have been used at a large UNIVAC 1108 installation known to two of the authors.

g) Static Deskchecking: A portion of code and its associated documentation can be fetched to the CRT screen. Retaining the same piece of code the superior levels of documentation can be inspected or the programmer may move upward or downward in the code/narrative stream. When a named procedure is encountered, he may indicate his desire to go out to look at that and then return to the point of call. Similarly, a reference to a data name can be used in an immediate inquiry (see c)) to elicit information on where and how stored (i.e., dimensioned, named field in a data structure), declared type, where else used, etc. Inquiry completed, the programmer can resume his perusal of the code. If this system is coupled to a device which can be quickly providing a hard copy facsimile of the CRT face, demands on the programmer's immediate

memory are reduced.

Programmer Scenario

The MP has received a request to perform maintenance on the "Colossus" system, which is the largest and most complex system at his installation. The request is extensive, involving modifications that affect input, output, and computation. The request documentation consists simply of a plastic card the size of a badge. He inserts the card in a badge reader connected to his console CRT unit and depresses a button on the console. There appears on the screen lines of textual information which indicate his name, a detailed breakdown of the manpower schedule for the modification, showing, where appropriate, the names of anyone else involved in the modification work, and a schedule for computer test time. He depresses another console button and receives a hard copy of the scheduling information which he inserts in his project workbook. By depressing another button on the console, the MP has displayed an abstract of the modification request. The abstract is organized in lines representing sentences, but uses abbreviations, symbols and suppresses unimportant words. By light penning the line he is interested in knowing more about the MP has displayed an expanded textual description of the change element indicated by the line. With a further touch of the light pen he gets displayed a flowchart of the system segments affected by the change described in the expanded text. He touches the light pen to a particular symbol on the flowchart and is presented with a display of "meta-code" that underlies the flowchart symbol. The "meta-code" is a shorthand notation of the actual source code that exists for that particular logic unit. Meta-code suppresses all coding that is only for the purpose of conforming to systems standards and displays a short-hand or abbreviated abstract of the logical portions of the code.

At this point the MP feels a need to understand more thoroughly the logic of the section he is examining so he depresses a button on the console that puts his unit in "programmed learning mode" (PLM). There appears on the screen lines of text indicating the options the MP has. One set of lines indicates how many "levels" of documentation exist for the particular logic segment he is interested in examining, and by light-penning he indicates the level he wishes to learn. (In this case the topmost.) Another set of lines indicates what types of documentation exist at each level (e.g., "flowcharts", "expando-text", "meta-code", "detailed source code", "object code", "all of above", etc.) Beside each documentation type appears a symbol denoting the status of that particular segment of

documentation (i.e., as in the previous scenario, whether the documentation is current, locked-out, etc.) He selects, with the light pen, "flowcharts". Frames of flowcharts appear on the screen in a rhythmic manner. The interval between frames can be slowed down, speeded up, or stopped by depressing the appropriate keys on the console or using a "joy stick". The MP can also, after halting a frame, request an "expansion" or "contraction" as described in the previous scenario. (Contraction would not be possible in this case since the MP started at the topmost level of documentation.) After the complete set of frames have been displayed for the MP, the programmed learning "option text" appears on the screen. The programmer selects, with the use of the light pen, the area of the documentation he has just seen that he desires to learn, and the level of comprehension he is interested in attaining (e.g., "full comprehension", "general systems logic", etc.) After this selection, the programmed learning text begins to be displayed on the CRT. The format of the responses to questions is multiple choice and the MP responds by selecting with the light pen one of the choices given. The programmed learning methodology is conventional; frames of questions and text appear, selection of answers are made, and the next frames to appear depend on the answer given. For wrong answers, re-enforcement routines are invoked. At the end of the programmed learning (occurring either by reaching the end of the course or by termination by the MP) a hard copy is created on the terminal printer that lists all of the areas where the programmer gave wrong answers, with possible implications of the wrong answer (e.g., "the TAPEX file must be prepared for input in proper sequence to the COMPUTEY routine or the system will assume the prior TAPEX file is the current data.") By further use of the programmed learning system the programmer familiarizes himself with those areas he feels he needs more knowledge in to perform the maintenance job. During the course of the modification task he may return to programmed learning to clarify areas he is uncertain of and to aid in understanding bugs or other unexpected aberrations in the system caused by the modifications.

After the initial programmed learning the MP is ready to begin the modifications tasks. First, he causes a display of the top level flowchart of the system to displayed and with the use of the light pen indicates those areas that will be affected by the modification. This causes the system to create a "test documentation file" containing that specified sub-set of the system. Next the programmer asks for a display of names of the data files that are associated with the modification-affected areas. From this list, with the use of the light pen, he can receive an "expansion" of any file which shows the format and

data elements available. From this list, with the use of the light pen, he can cause test data files to be created that contain copies of sub-sets of the existing files, and, optionally can receive a hard copy list of the files so created.

Next, the MP may request a list of all variable or data names with associated data types for any or all of the affected areas and may select all or part of the list for hard copy output. Similarly he may request a list of all I/O devices referred to.

When the MP is ready to begin his modifications he works with a combination of systems produced hard copy and the console CRT. Modifications are made first to the test documentation file beginning with the highest level of documentation and proceeding in sequence to lower levels. At each level and for each modification the MP has the option to "test" his changes at two levels. First, he may request a "syntax" test which will examine the modifications and display any errors in structure (e.g., "Logic segment COMPTAB has a two-way decision exit but only one exit is connected"). Next, he may request a "semantic" test which will examine the modifications and display any errors in content that the system is capable of detecting, (e.g., "variable DATAX does not change its initial value during the execution of COMPTAB.")

After the documentation test described above, the programmer requests the system to construct an executable program segment which it does from the information supplied previously regarding the modification affected areas and the associated data files. Then he enters on the console CRT the code modifications and requests tests of the various segments. During this phase he has the ability to "flag" certain logic segments and/or variable used by the system in order to generate dynamic traces which can be "played back" on the CRT.

After this check-out is completed he may request a "systems test". The system then integrates the changes into the larger resident systems test file and runs a full scale test which also, at MP option, will trace certain logic segments and/or variables and "play back" the various conditions of these at programmer option.

During the "playback" of a test the programmer may elect to insert "stops" in the code execution/simulation. These stops will cause a pause in the playback upon the satisfaction of pre-specified conditions such as the value of a variable attaining a certain range or a routine being entered. At the

time of such pauses the MP may elect to trace in greater or lesser detail the portions of the execution/simulation preceding or following the pause, or to retrieve certain data files, documentation test files or other portions of the system.

Systems Scenario

The structured documentation system is organized as separate primary files. Each such file represents a level of the structured documentation. Corollary files include texts for diagnostic routines, programmed learning and systems messages; and tables of pointers and file directories.

Processing operates under the control of an executive or handler routine that is similar to the one described in the previous scenario.

In addition there is a special "frame" handler that formats and presents to the CRT the randomly acquired frames of text and documentation that support the programmed learning and execution/simulation portions of the system.

A data formatting and extracting program operates the I/O test file portion of the system.

All of the above routines link to the various files through a separate "pointer" file organized in two major segments. One segment is for static pointers whose value does not change throughout the execution of the system. These include pointers that contain the addresses of production system documentation and data. The second segment contains pointers that change as the systems execution proceeds. These include the addresses of documentation test files and data test files.

Questions raised by the structured documentation scenario.

SD1 Will such a system optimize maintainability?

SD2 How will the "units" and "levels" of documentation be defined? As in the previous scenario, should such definition be subjective or objective?

SD3 How difficult will it be for a MP to learn to use the system? How will this offset any advantages of the system?

SD4 Can an effective algorithm be derived that will perform the text translation necessary for "expando" and "meta-code" techniques?

SD5 What about the hard copy "trade-off" alluded to in the preceding scenario? (It is not difficult to imagine a system configured in a way that the MP has to deal with greater volumes of more poorly organized hard copy than with present manual systems.)

SD6 What about the effects of out-of-date documentation referred to in the preceding scenario?

SD7 What hardware characteristics facilitate the SD documentation system?

## Part IV

## Interactive Aids Scenario

1. Dynamic Visualization:

Programmers frequently and laboriously combine the process of static deskchecking with a pencil-and-paper simulation of the program action upon typical settings of the inputs. This process is slow, prone to error and hindered by the need to constantly move back and forth between data representations (integer, floating point, octal, decimal, hollerith text, etc.). Alternate means for obtaining the same sorts of information are snapshot dumps, post-mortum dumps, trace routines and debug printouts inserted in the code. Each of these has its own virtues and drawbacks. The ability to elicit the dynamic behavior of a program or set of programs working together can and should be provided on an interactive basis. Two versions of such a facility are described below. Each has a proper role and one cannot be thought to be a complete substitute for the other.

a) Interpretive System: The interpretive system hews closely to the procedure followed in pencil-and-paper simulation with the advantages of being faster, more accurate and providing automatic conversion and formatting facilities. In concept, it is a direct descendent of single-step and breakpoint debugging practiced in the early days of computing. The specific implementation techniques as well as the operational modes and facilities to be provided are the objectives of research here, but some rough sketch of operational modes can be given by a description of a typical contemplated scenario. As regards this scenario, it should first be stated that the probable best use of an interpretive system is in the investigation of a

"relatively small" section of code, with "relatively few" inputs, parameters and data tables to be set up beforehand, and "relatively few" or "relatively minor" interactions with the rest of the system. In this context, then, the user single steps through the source code text of his program. The steps referred to are, by default, single source language statements but these are subject in some manner to flexible and instantaneous re-definition so that, for example, he may choose to look at the overall result of an IF-THEN-ELSE statement or go through each step of the data-determined clause which is actually executed. Or again, he may inspect the overall result of executing a procedure or choose to enter and step through its coding. At each step, he may pause and use inquiry facilities to inspect the current values of key items, returning after each such pause to the next step in the program flow. More sophisticated facilities which can be envisaged here are break-points keyed to program statements, FOR-loop indices, WHILE-DO or DO-UNTIL conditions and the alteration of prespecified variables.

Certain elements of this capability have been implemented in a prototype, FORTRAN--based system called GRAPE (Green,1970).

b) Playback System: A playback system (cf. Balzer, 1969), by contrast, is appropriate in the context of a larger, more complex section of code together with the data and files which support it. The gist of such a system is that a tape file is created which contains trace and dump information as the code is executed. This tape is later played back as a slow-motion movie on the CRT screen with the user able to speed up, slow down and jump forward or backward in the flow of action. The user is provided with extensive and flexible facilities for masking, filtering and summarizing the vast and otherwise unintelligible information content of the movie's data base. These facilities are exercised through such commands as--

- "Collect and display a table showing successive values of variable APRIME showing program module and line number where created."

- "Operate at high speed until ALPHA is greater than BETA then slow down for closer inspection."

- "Make a new film starting at APROG Line 1 and ending at BPROG Line 35 using GAMMA=0 and DELTA= -1.3"

- "Display the node and chord skeletal structure of the program providing identification of the nodes (branch points) and chords (linear code), and display at each chord the number of times it was executed (total or under specified conditions)", etc.

2. Project Control Aids:

We have noted elsewhere that when code is permanently altered, this can be tied to the associated documentation to inform management about the need for updating the documentation. There are a number of control aids with a more dynamic flavor which should be mentioned here.

a) Warranty Period Concept: One rough measure (and the worth of this measure should be investigated in the real-time context) of the solidity of a revised section of code is the number of times it has been executed. One use of the introspective system described above could be to provide management with such reports.

b) Programmer Performance and Progress: It can be assumed that no matter what the purpose of a console session is, the programmer will be required to identify himself and state the job control or project identification code for the work he is about to perform (see System Security below). This can easily be turned into a means for reporting to management the time being spent on the project. By also presenting the programmer with a set of "how did it turn out?" questions at the end of the session, progress can also be measured. A string of "I did fine" replies could be an indication that the programmer, at the very least, is deluding himself.

Questions raised by Interactive Aids Scenario.

Ial For an interpretive system what are the optimum dimensions (size) of the code and how many inputs are maximum for effective operation of the system?

IA2 For a playback system what is the definition of a "frame"? (Similar to the units definition question). What is the definition of "resolution" of the frames (i.e., homogenity of the

levels).

IA3 For a playback system how would frame synchronization between different resolution levels be achieved?

IA4 If the MP detects an error in the code while using these aids, how will he insert (patch) the code? Will the technique for error correction create non-integrated code sequences? ("Jumpy" frames in the playback context.)

IA5 In the context of these aids, what is the meaning of "recompilation"? (Obviously conventional recompilation will be extremely expensive). Is it possible to derive segmented interpretive compilers that perform structural integration with the whole complex of documentation and operational systems without, at the same time, requiring total multi-level recompilation?

IA6 What should the operational definition of "debugged" be? Can statistical measures be derived that will determine when a program's reliability achieves a predetermined value?

IA7 What techniques, existing or new, should be used to achieve system security? What problems, such as interference between programmers, will such techniques create?

IA8 In the real-time environment how should the question of "resource-ownership", be resolved? How does this question affect interfaces with the MP as he performs maintenance modifications?

## Part V

## Testing Scenario

### Programmer Scenario

The MP has at his disposal a Systems Test Language (STL) which he uses when performing tests. This language enables him to request information specific to the test problem, to examine and alter test file segments and to automatically compare test results with prior results or with production generated results.

The STL corresponds to the structured documentation and HOL networks in that it allows the MP to communicate with any logical unit on any level of the structured system.

Specific STL facilities at his command are:

a) Data examination. Allows the MP to request a "picture" of any data element resident in the test files. Requests can be conditional, dependent on the range of value of a variable or upon the entry of a particular logic segment.

b) Documentation/Code examination. Allows the MP to examine any unit segment of code or related documentation upon the satisfaction of prespecified conditions during execution.

c) Data/code/documentation Comparisons. Allows the MP to compare data results or upon the satisfaction of prespecified conditions, to compare source code and related documentation in the before/after modification mode.

Systems Scenario

The STL system operates through an interpretive routine that examines the STL commands from the MP's console and translates them into an interpretive string notational language. The string language is composed of functional designators (such as "retrieve", "insert", "compare", etc.) address parameters related to the designators (specifying such things as the address limits of data or documentation to retrieved) and pointers that specify logical connectives between, for example, different levels of code and documentation in the structured system.

Questions raised by Testing Scenario.

TP1 What are the smallest units of data/documentation that should be accessible to the MP through the use of examination commands? What are the trade-offs between unit size and MP effectiveness?

TP2 Should STL logic be imbedded in production systems in order to facilitate rapid execution or should it be imposed at test time?

TP3 What interface problems will occur between STL and well structured documentation and code? What provision should be made in well structured systems to accommodate an STL language?

TP4 Aside from comparisons, what arithmetic routines, if any, should be available to the MP?

PART VI

# Position Paper: Higher Order Languages and Maintainability

## 1. Introduction

An underlying assumption of this paper is that we are dealing with the maintenance of large software systems which have been written in an HOL. This assumption breaks down into two logical parts: first, consideration must be given to those features of an HOL which make it theoretically acceptable as a vehicle for scripting the components of a large software system; second, consideration must be given to those features of an HOL which make it acceptable practically as a vehicle for constructing and maintaining such a system.

The first of these considerations speaks for itself; the second requires some preliminary elucidation; both will be dealt with at greater length in what follows.

There are two points to be made concerning the second consideration stated above. The first point is to note that (original) construction and maintenance are often lumped together despite indications in the data gathering phase of this study (e.g., BUIC Case Studies in Appendix B) that there are inherent differences in the two processes which then presumably call for different sets of tools and aids. In our view, the approach which allows a unified approach to these two processes is the content of the second point: one must regard an HOL as being a part of a total computing environment. Thus, the second consideration should be rephrased as:

> "second, consideration must be given to those features of a total computing environment and to those features of an HOL operating within that environment which make it acceptable practically as a vehicle for constructing and maintaining such a system."

In Section 2 below, we shall examine the first of the above considerations, namely HOL features providing amenability for writing large software systems, and we shall cite relevant cases where HOL's have been so used. In Section 3, we shall examine the second of the above considerations as rephrased. This will consist of an examination of the foregoing scenarios to isolate their implications for HOL's. Next we shall examine current experimental work which bears promise of providing a total

computing environment and an HOL to enhance construction and maintenance of large software systems, and finally, we shall examine briefly language features of present HOL's that are intended to enhance maintainability or which have unintentional positive or negative effects upon maintainability.

In Section 4, we will look at an area of endeavor generally regarded as being out of the mainstream by workers and writers [e.g., Cheatham, forthcoming; Cocke and Schwartz, 1970] in the area of software, HOL's and compilers. This area has to do with the closer integration of hardware design (computer architecture) with HOL and computing environment design in order to reduce the amount and complexity of the requisite software. Examples abound in this area and we shall review these.

In Section 5, we present a series of questions on HOL's which we feel need to be answered in the future to assure that total computing environments and their HOL's provide optimal maintainability of the software systems created using them.

## 2. Writing Software Systems in an HOL

One of the driving forces in the evolution of HOL's cited by Cheatham (1971) is the need for

> "languages in which programs for a certain class of problems can be written 'naturally' . . ."

What this means is that the programmer requires a language which provides him with the data types and structures and associated operations which are natural to the problem area. Using a more general purpose programming language, he is forced to build these natural entities out of the more primitive atoms and operations provided by the general purpose programming language often in forced, contrived and artificial ways. Thus, says Cheatham

> "The argument that most any programming language is theoretically adequate for most any programming task just does not make it an acceptable vehicle in practice."

We note that there is a lesson to be gleaned here for the problem of software maintainability. One has the intuitive feeling that the learnability (and therefore the maintainability) of a software system is enhanced by a natural association between that which is being manipulated and the prescriptions of manipulation (programs). Conversely, the more layers of artificial

build-up required to construct the natural entities to be manipulated, the more difficult it is for a novice to learn what is going on. Such an intuitive feeling should be subject to more precise statement and subjected to rigorous measurement.

The past decade has seen the speed of hardware increase and the unit cost of hardware decrease by many orders of magnitude. The costs and speed of software production and maintenance have experienced no such favorable changes. By contrast, the production of application programs in many areas has been speeded and made less costly by the introduction of HOL's to replace assembly code (which in turn was a vast improvement over absolute coding). Thus it is natural to try to apply this lesson to the problems of constructing large software systems. But, as stated by Cheatham:

> "One application area which still suffers from the lack of good programming tools, or perhaps more appropriately, from an improper attitude about [and knowledge of] the tools which do exist, is that of large system construction."

The problems encountered in thus using an HOL are those cited above, namely a lack of proper data types, data structures and operations. To these should be added, as regards this area, a paucity of flexible control structures.

Another factor leading to the unacceptability of HOL's in this area is that too often the object code produced has been too lengthy and/or has run too long. This is, in part and in our view, a result of the category of problems cited above: the natural entities manipulated by a large software system are hardware registers (etc.), interrupts and individual bits (amongst others). An HOL which does not provide convenient means for handling these is bound to provide inferior results. Another part of the object code length/time problem, in our view, is insufficient thought given to integrating the HOL design with the design of the computer architecture. This will be discussed at greater length in Section 4.

A minimal list of HOL attributes to provide a vehicle for writing compilers is given by Cocke and Schwartz (1970, p. 253). Another such list is given by Peschke (1971) as proposed improvements to PL/I. The two lists overlap in parts. The first list is given as Table E-1.

TABLE E-1

HOL Features for a Language to Write Compilers[1]

(i) Efficient access to machine part words as variables in the language. This is very important if the compiler is to be able to use densely packed tables.

(ii) Data structures and allocation rules which permit control over placement of variables in memory including control of overlays, and which permit the combination of heterogeneous variable types in a single structure or entry (as in a symbol table).

(iii) Some form of based storage permitting convenient shifting of tables in core and allowing table structures to be extended to newly allocated blocks of core.

(iv) Name-scoping rules permitting easy combination of separate routines written by different people.

(v) Efficient and flexible calling sequences. In this connection, some of the ideas on subroutine linkage optimization discussed in a later chapter may be valuable.

(vi) Access to all machine instructions, hopefully in a form which does not obstruct global optimization of the compiler code. One way of providing this access is in terms of a package of subroutines.

(vii) At least a rudimentary system of macros permitting conditional compilation of the system source language should be provided. Such a tool is useful for a variety of purposes, including isolation of compiler parameters, avoidance of repeated and error-prone insertion of repetitive code blocks, and production of a number of slightly variant versions of a compiler.

(viii) It must be possible to initialize variables, and, in this connection, provision must be made for the convenient treatment of character and bit string constants.

(ix) Recursive routines are useful for the expression of a number of compiler processes.

---

[1]Cocke and Schwartz, 1970.

To the list given in this table should be added a wider variety of control structures including the ability to create, start, coordinate, stop and destroy cooperating asynchronous processes; coroutines, etc.; and access to and control over internal and external interrupts. These things are usually hidden from the HOL user.

The approach and philosophy adopted by the ECL group at Harvard (Wegbreit, 1971) is twofold:

- the programmer is not to be denied access to the key points of system control, those things which we noted above are usually concealed from the HOL user.

- one cannot possibly guess the kinds of data, data structures, operations, control structures, etc. that the HOL user is going to want. Therefore, instead of giving him what we (ECL group) think he should want, we will provide him with an easily used capability for extending the language to suit his needs.

One cannot but wonder what the trade-off in maintainability will be of the gain in naturalness versus the possible need to learn new programmer-defined language constructs in such an extensible language environment.

This section would not be complete without reference to major software systems which have been written in an HOL. One outstanding example is provided by the Burroughs 5000 (5500/5700) computers and their higher numbered big brothers. Their entire software (Master Control Program, compilers) is written in a family of specialized dialects of ALGOL 60 extended in many of the ways indicated above to put the HOL user closer to the hardware. It should also be mentioned (jumping the gun on Section 4) that the architecture of these computers also puts the hardware closer to the HOL.

Other examples (Cheatham, forthcoming) are MULTICS which is 90% (figure supplied by E. Fredkin, personal communication) written in PL/I. [One wonders what the relation of that figure is to Peschke's (1971) suggested improvements to PL/I or if Peschke is concerned with the same PL/I implementation as was used in MULTICS.] Cheatham also mentions an operating system for the Honeywell 638 using FORTRAN and a logistics system for the USAF Logistics Systems Command using COBOL. Cocke and Schwartz (1970) mention that IBM/360 FORTRAN H was written in an extended FORTRAN, but no indication is given of the nature of the extensions.

## 3. Features of HOL's and a Total Computing Environment that Enhance Maintainability

In parts I-V of the present appendix scenarios from an ideal future computing environment have been sketched. The aspect of these scenarios that impinges most directly upon HOL's is the Interactive Aids Scenario wherein we see a programmer "stepping through" the (HOL) source representation of his program. By "stepping through" we mean that each statement is being executed interpretively acting on programmer-specified data. Parts of the program such as procedures whose correctness is either assured or of no present worry to the programmer are present in compiled, object code form. When the programmer finds the error he has been seeking or the proper place(s) to make a desired modification, he makes the change from the console and again executes interpretively. When he is satisfied with the results of the execution, he may then order (from the console) the compilation of the code segment and its placement in the proper program file.

The ECL programming system and its associated HOL, EL1 (Wegbreit, 1971), appear to us to capture many of the capabilities of an HOL and a total computing environment mentioned in our scenarios or implied therein. The avowed goal of ECL is "an environment which will significantly facilitate the production of programs." The measures taken to reach these goals are

1) It (EL1) has been designed to be used in an interactive environment.

2) EL1 can be executed interpretively or compiled and interpreted and compiled code segments can "be freely intermixed with no restrictions."

3) Execution can be suspended while the programmer "examines data or program, modify either, and optionally resume."

4) A variable can be made "sensitive" and while in that state "changes to its value are monitored and an interrupt generated whenever a (programmer-specified) predicate [associated] with the variable becomes true."

We turn now from this consideration of the ideal and ECL's approach thereto to a consideration of more mundane, detailed features of current HOL's which either are intentionally designed to enhance maintainability or having other intended purposes, exercise, nonetheless, a positive or negative effect on the maintainability of software generated using them.

## 3.1 Intentional Maintenance Features

a) Compile-time DEFINE facilities for text substitution or macro-expansion. Examples: Burroughs ALGOL 60, FORTRAN V.

b) FORTRAN V PARAMETER statement which can be regarded as a special example of a).

c) Conditional compilation directives. Example: DELETE statement of FORTRAN V.

d) The ability to name and subsequently include sections of code or data base definition by reference to said name. This centralizes the locus of future maintenance effort as when, for example, a given named COMMON block must appear in many FORTRAN subroutines. This may also be regarded as a special case of a). Example: INCLUDE statement of FORTRAN V.

e) The use of the ON statement group as in PL/I for conditional debugging output.

f) The ability to request the construction of cross-references and other aids at compile time.

g) The information value of compiler diagnostics and the strategies employed to overlook errors and provide a rationale interpretation.

One aim of future research in the area of HOL's and maintainability of software should surely be to expand and refine this list especially in the light of radical innovations such as ECL. An example of such a feature is suggested by Knuth (1970): generate statistics on the frequency of execution of each statement. Then areas of high frequency (e.g., loops) can be concentrated on for the most cost-effective application of optimization techniques.

## 3.2 Other HOL Features with Maintenance Implications

a) The GOTO statement. The Mills (forthcoming) - Dijkstra (1969) approach to structured programming or constructive proof of correctness calls for HOL's lacking the GOTO statement so that every program atom can only be entered at the top and left at the bottom. It is not

clear to us that a program optimally constructed for proof of correctness is also optimally maintainable and this is the subject matter for a proposed study elsewhere in this report.

b) Other Control Structures. We have established elsewhere by literature reference and interviews that one of the most desirable tools needed by a maintenance programmer is one enabling him to gain dynamic insight (visualization) of the behavior of his program. Control structures such as recursion, coroutines, independent and cooperating parallel asynchronous processes will require special efforts to provide dynamic pictoralization. Console imperatives that build time-slice tables will surely be required.

c) Source Input Structure. Existing HOL's exhibit two types of source input structure. the FORTRAN type of structure generally calls for a main program and zero or more separately compiled subprograms. (UNIVAC's FORTRAN V provides a minor partial exception in this regard.) The need for globally defined data base items and arrays in this environment gives rise to features such as blank or named COMMON sections and BLOCKDATA program elements. Language features such as the ability to name subsequently included sections are added to alleviate the maintenance problems engendered by the appearance of a given COMMON block in myriad subprograms. An intuitive feeling about this approach to source input structure is that for a given size of program, the "distance" (however this is eventually defined and measured) of distant referents is increased.

By contrast, the ALGOL (60) type of source input structure calls for one program with all of its subprograms (except for library programs) contained within it. The scope of data definitions is implied by the nesting of the domains of definition in the source order. An intuitive feeling is that this type of structure has beneficial effects as regards "distance" of distant referents, but a simple change to a small part of such a program requires a recompilation of the total program, whereas the former approach requires only that portion of the program which is in error to be recompiled.

It is now apparent that block structured HOL's and separate recompilation are not incompatible. Languages

exist having both. The ECL model may make the argument old fashioned.

d) Method of HOL Implementation. In general HOL's may be implemented in one or a combination of three distinct ways:

--Interpretatively: The source text is scanned and executed by an interpreter. Examples: APL, ELl.

--Compilation: Compilers are, of course, common.

--Hardware or Microprogram Implementation: The source text is translated and re-ordered in a vastly more simple and quicker process than in full compilation. The result of this process is still relatively close to source form and is directly executable in the hardware, or by microprogrammed interpretation. Examples are SPL on the SPLM and SYMBOL on SYMBOL (Section 4).

As noted above the ELl language (Wegbreit, 1971) being developed at Harvard allows the programmer the choice of either interpretation or compilation; and, moreover, the compiler is callable at object time by the programmer. The compile time, load time, object time distinction disappears.

e) Declarations. Were one asked to make an intuitive guess about the types of corrections maintenance programmers actually make [à la Knuth (1970)], mistakes involving declarations would rank high in the answer. HOL's exhibit a range from nothing need be declared (APL, SYMBOL) through only some things need be declared (FORTRAN) to everything must be declared (ALGOL). Some thought needs to be devoted to discovering the true mistake generation statistics of declarations (and other language features also). Some optimal mix between the extremes may exist and this mix should be searched for and its contextual parameters better understood.

g) Identifiers. Another intuitive feeling concerns the length of identifiers: the longer the better, because the longer they are the better they convey the intent of the original programmer. This is surely subject to some exact measurement.

## 4. Hardware as a Replacement for Software

There is considerable evidence on hand that better computing systems can be designed if software people work together with engineers. This was the approach taken in the design of the Burroughs machines alluded to above. There is evidence that this idea may catch on. For example work now completed or in progress includes such projects as follow:

> Contract F04701-70C-0065 (SAMSO), defined SPL/Mark III which is a language for an on-board missile guidance computer (real-time process control). Simultaneously a compatible, stack-oriented computer was developed (Advanced Guidance Computer - AGC). Programming studies showed that the AGC exhibited a 60% reduction in space requirements over a conventional single address (ATS) architecture on a set of guidance-oriented benchmark equations. The AGC code was produced by hand simulation of a rather inelegant compiler. Timing comparisions are not yet available, but the AGC code need not be faster than the ATS code as long as it meets the timing tolerances of the application.
>
> Contract F04701-71-C-0200 (SAMSO) is an architecture study (SPLM) for a direct execution processor (SPL/Mark IV - direct execution version). This is a processor in which the source code is slightly rearranged and reformatted by a loader and the output of the loader is directly executable. This output corresponds roughly to the sets of tables and syntax trees produced by a conventional compiler prior to code generation. There is no fixed word length and pre-execution binding and storage allocation are at a bare minimum. Using context-dependent Huffman coding techniques another 20% space reduction can be achieved.

The recently developed SYMBOL language and computer is an example of this concept at work [four sequential papers, SJCC 1971 of which Chesley and Smith (1971) is the first]. The first SYMBOL machine has been delivered to Iowa State University which expects diminished operating costs partly attributable to almost no need for software maintenance: the whole compiler and operating system are hurd wired. One will surely want to keep an eye on this to see what their actual experience turns out to be. (For a brief summary description of SYMBOL see Computer Design, April, 1971.)

## 5. Questions for Future Investigation

HL1 What is the optimum (from the standpoint of MP effectiveness) length for labels? What are the trade-offs generated by label length between programmer effectiveness and compiler efficiency?

HL2 What is the "optimum" mix of direct hardware implementation, software interpretation and compiled object code in implementing an HOL? Which language features are best handled in the ways cited? Can and should the programmer be able to specify the mode of implementation? What are the trade-offs between MP effectiveness and compiler efficiency?

HL3 Can (should) the HOL language be structured? If so, what problems are raised in progressing between levels when maintenance is to be performed? How would multi-level testing be handled (i.e., segmented recompilation vs. total systems regeneration).

HL4 What systems will be necessary to interface maintainable HOLs (structured or conventional) with structured documentation systems? How will modifications to one or the other be handled?

HL5 Should the lowest system language level for the HOL be microprogramming structured? If so, what advantages, disadvantages and trade-offs result?

HL6 What hardware features should be specified for the maintainable HOL? How are these features related to application requirements?

HL7 What real time application considerations are raised by a maintainable HOL? How will the HOL interface with operating systems and resource ownership algorithms? What maintenance problems will message/transaction handling interfaces raise?

## References


Balzer, R. M. "EXDAMS - Extendable Debugging and Monitoring System." Proceedings, Spring Joint Computer Conf., 1969.

Cheatham, Jr., T. E. "The Recent Evolution of Programming Languages," to be presented at the IFIP Congress; Ljubjana, Yugoslavia, August 23-28, 1971.

Chesley, G. D. and Smith, W. F. "The Hardware-Implemented High Level Machine Language for SYMBOL." Proceedings, Spring Joint Computer Conf., AFIPS, pp. 563-573, 1971.

Cocke, J. and Schwartz, J. T. Programming Languages and Their Compilers, Preliminary Notes, Courant Institute, New York University, April 1970.

Dijkstra, E. W. Notes on Structured Programming, Technische Mogeschool Eindhoven, 1969.

Green, J. "Program Analysis-A Problem in Man-Computer Communications." NASA Technical Report NASA TR R-338, June 1970.

Knuth, D. E. An Empirical Study of Fortran Programs, Stanford U. Computer Science Department, Report #CS-186, 1971.

Mills, H. D. "Syntax-Directed Documentation." Comm. ACM, 13, No. 4, April 1970.

Peschke, J. V. "PL/1 Subsets for Software Writing," SIGPLAN Notices, Vol 6, No. 4, May 1971, pp. 16-22.

Wegbreit, B. The ECL Programming System, Division of Engineering and Applied Physics, Harvard University, Cambridge, Mass., 1971.

# APPENDIX VII

## Technical Approach and Aims

## for a

## Path Analysis Feasibility Study

## 1. Detailed Procedure

The experimental study will be done according to the following detailed procedure:

1.1 Select a program, with associated documentation, for modification. The program selected will have been written in a higher-order-language, probably FORTRAN. The documentation will include a program narrative, a high level flowchart indicating the logical relationship between the various program segments or modules, a detailed flowchart indicating the logical relationship between program statements, and a diagram of input/output formats and contents. (See Attachment.) The program itself will be relatively short in terms of statements (about 100). It will provide a computational solution to a non-trivial problem, e.g., a matrix inversion algorithm, an exponential smoothing routine, a curve-fitting routine, etc.

1.2 Specify a modification to the program. The modification will be clearly described in narrative form, but such description will pertain only to the functions of the computation to be modified, rather than to the language and its particular configurations in the program. The modification will be non-trivial and will require changes in the program that will affect several logical modules or segments.

1.3 An experienced, knowledgeable programmer makes the modification. A member of the project team, who is familiar with the program and the application, will make the modification and record the time it takes to make it. While doing so, he will describe into a tape recorder the sequence of steps that he takes in performing the modification. The entire project staff will then examine the modification steps to derive the minimum path to successful completion of the modification.

1.4 A "naive" programmer will make the modification. (Note: the modification will only be made to the level of changes to the source code. No computer testing will be involved.) The programmer will be naive in the sense that he will not be familiar with the program, but will have a working knowledge of the application. He will first read the narratives that describe the program and the modification and will be asked to make a time estimate for completing the modification. This estimate will be "negotiated" with the project staff, using the original programmer's time as a standard, until a commonly agreed-to time is derived. (This is intended to simulate the "real" environment of a maintenance programmer in his interactions with his management.) While making the modification the programmer will have available all of the documentation and code for the program and the narrative specifying the modification. He will also be allowed to ask specific questions of the experienced programmer and will receive specific responses. These may be filtered through the study director to make sure that the experienced programmer reveals only technical details and not his general approach or path. (The interrogation process will simulate the resource of consultation with more experienced programmers.) While performing the modifications he will also describe into a tape recorder the sequence of steps he is taking and the reasons for taking each. This spoken narrative will be done according to a specific format derived by the project team and supplied to the naive programmer. (See Appendix.)

1.5 After the naive programmer has completed the modification, the project team will analyze the recorded tapes and the modification. A table will be prepared that shows the steps the programmer took with associated reasons for those steps. The table will separate steps that are on the minimum path from those that aren't.

2.0 Results Expected

2.1 The recorded tapes and tables described in 1.5.

2.2 A written report, by the project staff, that will present

(a) explanations of the deviations from the minimum path,

(b) evaluation of the various reference resources available to the programmer,

(c) suggestions for specific remedies to the path obstacles encountered, and

(d) suggestions for further research based on the findings (feasibility recommendation).

## 3.0 Application of the Results

3.1 Deviations from the minimum path. The reasons for such deviations may provide us with new insights into the fundamental factors that inhibit the effectiveness of maintenance programmers. From such reasons it may, for example, be possible to deduce human characteristics that are incompatible with the present way of performing maintenance programming, and to suggest ways in which the working environment may be restructured to accommodate these characteristics.

3.2 Evaluation of reference resources. This may provide indications of how reference resources might be improved. It may also provide ideas for further research into new ways of structuring reference material or entirely new materials that would be more conducive to the effectiveness of the programmer.

3.3 Transcripts, with staff comments and analysis, of the tape recordings. These could provide a basis for future studies. For example, if new reference materials were to be tested, a similar experiment could be conducted using the new materials and a transcription of that experiment compared with the first to detect differences.

3.4 Suggestions for Further Research. Based on the findings in this study, recommendations concerning a Path Analysis research project will be made. The recommendation will be supported by analysis of the findings to determine the areas that would have to be studied and further defined. Each of these areas will be analyzed to determine

(a) the feasibility of providing the answers to the questions raised. (For example, is it possible to define a "path" through a program with enough accuracy to support research conclusions? and if so, what method might be used?),

(b) the resources and time required to conduct such a project,

(c) the detailed procedures that might be used, and

(d) the results that might be expected.

Based on these points, an estimate of feasibility and cost effectiveness will be made that should indicate whether further research is warranted.

ATTACHMENT

1. Program Preparation

The programming material will consist of

(a) the source code,

(b) a detailed flowchart,

(c) a top level flowchart,

(d) a program narrative, and

(e) I/O format diagrams.

Each of these materials will be segmented, by marking out with a felt pen, and each segment will be numbered. (For example, the detailed flowchart may be numbered "2", and the segments of the detailed flowchart numbered 2-1, 2-2, etc.) The segments will be selected in a way that they represent, to some extent, a logical entity. This segmenting will be arbitrary in that it is not expected to be unique. The correspondence between segments in various levels of documentation will deliberately not correspond on a one-to-one basis. The program material will be prepared in this manner so that the programmers may communicate which program reference area they are referring to as they progress through the modification effort.

2. Outline of questions the programmer will have before him and try to answer as he goes from one reference segment to another (path).

2.1 What new knowledge do you expect to acquire by looking at this segment? (If no action is to be taken.)

2.2 Why was this segment selected over any others to provide this knowledge?

2.3 (Before looking at the segment.) What is your concept, at this point, of the actions that will have to be taken to effect the modification?

2.4 (After looking at the segment.) Same question as 2.3.

2.5 Was looking at this particular segment helpful (clarifying) or hindering (confusing)? Why?

2.6 (If the programmer is taking action, i.e., modifying code.) Describe the action you are taking. Why are you taking it? How do you expect to test its accuracy? What most recent reference segment caused you to take this action (may be more than one or none)?

# APPENDIX VIII

## Instructions for Path Analysis

## Experimental Programmer

You are participating in a study wherein you will be asked to make a modification to a program. You will only make the modification to the source code. You will not compile or test. However, if your modification is incorrect you may be asked to continue working on it, much as you might if you had gotten incorrect results from a computer test. At all times you should strive to perform as you would in a normal programming working environment.

The purpose of the study is to discern the paths you take through the program and its associated documentation while attempting to make the modification. You should follow your normal methods, as much as possible, and attempt to make the modification within the time you estimated. You have been provided with the following items:

1. Program materials; coding pads, source listing, detailed flowcharts, top-level flowcharts, program narrative, a narrative description of the modification you are to make, and blank paper and writing instruments.
2. A timer.
3. A tape recorder.
4. A path log.
5. List of questions.

You should proceed with the modification following these procedures:

1. Keep the timer running whenever you are doing work connected with making the modification, including discussions with the program consultant. Shut the timer off when you are performing functions connected with the path analysis study such as writing in the log or answering questions into the recorder.
2. The elapsed time shown on the timer should be compared with your estimated time to perform the modification. Every attempt should be made to complete the modification within the estimated time.
3. Each time you refer to the program source listing or associated documentation enter the code identification of the documentation section on the path log before you read any part of the section.

   Next, (and before reading the section):
4. Answer the questions, speaking into the recorder, identified as "before" on the question sheet.
5. After you have finished with the section you are referring to, and before doing anything else:
   a. Answer the "after questions into the recorder.
   b. Record the time you have spent referring to the section on the path log.

6. For any other activity related to the modification effort, record on the path log according to the code at the bottom of the log. Then follow steps (4) and (5) for that activity.

During the modification effort you may ask questions of a "program-consultant". The questions must be specific and clarifying only. (In other words, questions seeking techniques for making the modification are not allowed.) All questions must be submitted in writing (stop the timer while writing questions) and will be responded to verbally (timer running). Questions may be disallowed if they are felt to be too general or if the answer would be too directive in giving modification techniques.

## Questions to be Answered into Recorder

BEFORE reading a new segment of documentation:

1. Give the code identification of the segment you are about to read.
2. Describe in detail the knowledge you expect to acquire by looking at this segment.
3. Why did you select this segment, over any other, to provide this knowledge?
4. What is your concept, at this point, of the actions that will have to be taken to effect the modification?

AFTER reading a segment (before going to another):

1. Give the code identification of the segment you just read.
2. What is your concept, at this point, of the actions that will have to be taken to effect the modification?
3. Was looking at this particular segment helpful (clarifying) or hindering (confusing)? Why?

If you are MODIFYING SOURCE CODE:

1. Describe in detail the modification you are making.
2. Why are you making this modification?
3. How do you expect to test its accuracy?
4. What reference segment or segments caused you to take this action?

| | Path Log | | | Timer | |
|---|---|---|---|---|---|
| | Enter ID | Enter Elapsed Time | Use Recorder | Off | On |
| BEFORE reading a new section of documentation or other actions (A.1, A.2 or A.3) | X | | X | X | |
| AFTER reading a section of documentation or taking other actions. | X | X | X | X | |
| WHILE reading a section of documentation or taking other action. | | | | | X |
| WHILE writing questions for programmer consultant | | | | X | |
| WHILE programmer consultant is answering questions. | | | | | X |

Programmer Activity Table

Path Analysis Study

# APPENDIX IX

## Specification of Modifications for Path Analysis

## Modification #1

Specification:

The seat restriction when applied to "LNCH" (lunch) is ridiculous and should be removed. Note, the only way you can create a Hollerith constant in this variety of FORTRAN is to read it in.

## Modification #2

Specification:

There is an error in Program II which results in an abnormal program termination. A deck of the run which evoked this error is provided with a printout containing some useful clues.

## Modification #3

Specification:

You will note that the MCSF contains a field which tells how many sections there are of each course. This figure is presently generated in a cumbersome manner. You will revise the scheduling algorithm to take advantage of this datum.

## Modification #4

Specification:

There is a similar error in Program III, i.e., a case when an array subscript can go out of bounds. Although no run has been made to test this error, you will assume that if such a run were made that errors would take the form:

THE LINES

```
IF (JARR(KZ) - CODSY) 67, 62, 67
```

and

```
IF (SVCOD (IZ, JZ) - CODSY) 70, 68, 70
```

yield erroneous comparisons

# APPENDIX X

## Minimum Paths According to Staff Programmer

Modification 1 Minimum Path

Time: 26 minutes of which 20 minutes were reading entire documentation, more or less sequentially.

1. Modification Narrative.
2. Source Code. Scanning through to look for variable with word SEAT in it.
3. Page 10: Disk File Lay Out. Discovered that variable SEAT contains the number of seats in a section matches with the seat restriction.
4. Source Code. Scanning through looking for a statement where ISEAT is decremented. Found it at line 3010 which reads ISEAT = ISEAT - 1. Noted that it occurs in a DO-loop.
5. At this point speculated that the modification required would be to jump to the end of the DO-loop if name of course is LNCH. This modification would be inserted before line 3010 in the source code.
6. List of variables. Scanning through looking for the name of the courses.
7. Page 10. Looking for the format of masters course schedule file. Found that CAT is the course or catalogue number.

8. Made the modification. Got around the Hollerith problem by changing the blank constant card to read "LNCh". Read that constant into a place defined as "EAT".

Modification 2 Minimum Path

Time: 24 minutes

1. Statement of Modification.
2. Run termination information. Discovered that the error occurred at or after the first two names. The second name was output correctly and the third one was suppressed.
3. Data cards for second and third names. Looked at the cards and noticed that the second name appeared to be O.K. but that the third had 15 requests.
4. Page 8. Narrative. Discovered there is a maximum of 14 requests per student allowed. Deduced at this point that an improper test for 14 requests is occurring in the code.
5. Source code. Page 55. Lines 1040 through 1070. Scanning through to find dimensions that are equal to 14. Discovered that there are three and only three variables so dimensioned. Scanned the entire code for occurrences of these names and for unusual uses thereof.
6. Source code. In scanning that code the variable names that are contained in DO statements with a range from 1-14 and initialized to 0 are eliminated from consideration.
7. Source code. Line 1440. A test here occurs for $N \geq 15$. This causes a new direction in searching for the proper modification, i.e., looking for uses of N since N is a subscript affecting variables that have the $N \leq 14$

restriction. Discovered that immediately below line 1440 is a DO for a range of 1 to N using a variable NSECT that was one of those variables with the 14 restriction on it. Discovered that line 1630 is where DO loop branches if $N \geq 15$. However, it seems that the test that the program makes preserves the condition of $\leq 14$.

8. Run printouts. Look at the error message which indicates that subscripts overflow destroys a table.
9. Listing of variables allocations. Discover that the first variable in the list is dimensioned 14!
10. Source code line 1430 in 1360 where the beginning of a routine occurs that is to process request cards at first card N = 0. Thus N = 14 for the 15th request card, the test will fail, and the subscript will overflow.
11. Making the modification. Changing line 1440 to test for N versus 14 instead of N versus 15.

Modification 3 Minimum Path

Time: 13 minutes.

1. Statement of modification.
2. Page 10, Figure 5. Layout of master course schedule file. Discover here that the number of sections is called "ISEQ".
3. Source code page 58 program 3. Scanning through looking for places where the number of sections (ISEQ) is being computed. Discovers:

   At lines 2480 and 2490 ISEQ is read, however, it is read just before the program termination so this is eliminated as a place where it is computed.

   Lines 2970 and 2980 are the only other places where ISEQ is read. Following down from there at line 2500 it is discovered that ISEQ is computed from IREC 2 and IREC.
4. Source code. Lines 2420 and 2430 read IREC, and IREC 2. At this point, from prior modifications, it is remembered where and how IREC 2 and IREC are used. On the basis of this a deduction is made; that to make a modification the second read should be eliminated because you should be able to get the number of sections after reading the file.
5. Source code. Line 2450. Here it is seen that IREC is altered under certain conditions. Therefore, it will be necessary to concentrate on a new way of computing IREC 2 since IREC is altered after it is read.

6. Modify the code. Modification consists in constructing a new variable IREC 0 and setting IREC = the new variable. Then replacing line 2500 by the statement NOSEC (L) = ISEQ - IREC.

Modification 4 Minimum Path

Time: 18 minutes.

1. Statement of Modification.
2. Memory Map. Looking here to see where CODSY is used. Looking for the address of the variable. Discover that SVSEC is on right side of CODSY. Note its location. Suspect overflow is wiping out next word. Discover that SVCOD is to left. Wonder is CODSY is being wiped out by over or underflow of SVCOD or SVSEC.
3. Source code. Scanning through to find DO loops where SVCOD and SVSEC are used. Since they are arrays assume that they must be in a DO or I/O statement. Scanning through the code and through DO loops in top to botton sequence in the code looking for DO loops where one of the bounds is a variable. Find a suspect at statement 67 which reads LM = L - 1.
4. Source code. Scanning through looking for a situation where you can get to statement 67 with $L \leq 1$. Discover that situation in statement 51 where L ranges from 1 to 14. So in looking for the place where L = 1.
5. Source code. Eliminate the first DO since $L \neq 1$. Scan for another DO loop and find one just below statement 73 that matches, i.e., statement 460 where L = 1 branches to the DO.

6. Modifying the code. Insert between statements 65 and 73 an IF statement that tests for L = 1 and branch accordingly.

# APPENDIX XI

## Experimental Programmer's Verbalization While Making Modifications

## Introduction

The following is a typed transcription of the tape recorded comments made by the experimental programmer, Jan Overton, as she worked on the modification.

## Quotations

Time: 10:31 A.M.

Modification #1.

Just opening the documentation to survey it, to get a general idea of what the program is about, and to just generally read through and find out what the program is about.

I have read the documentation to page 19. The time is 11:40 A.M. A concept, at this point, of the actions that will have to be taken to effect the modification, is: either to increase the seat capacity in the course called Lunch to four positions, and enter all nines (which would essentially give ten thousand seats to Lunch); or, each time in the scheduling program, before the Seat field is tested, test to see if we are trying to schedule a Lunch period, and then by-pass the test to see if all the seats are filled.

Time: 1:07 P.M.

I am continuing to generally read through the documentation starting with page 20.

I have just finished reading the documentation through page 61. I am now ready to make the modification. What I will do is change the seat specification on the data input card for the one named Lunch, and start with a low negative number like -1. Then in the program each time before Seat is tested, I will test to see if Seat is already negative; this is the only "class" in which "Seat" should be negative; and it should always be negative here.

I am going back in the documentation to the card input layout, the master course card, on page 7. Now I am going in the documentation to page 60 and 61.

I have made the modification. I simply changed two statements, statement 2530SCHD and statement 2630SCHD; and I tested Seat, and instead of branching to the class-full condition if the seat field was negative, I branched there only if it's zero. If it's either negative or positive I branched to the condition in which there are still seats left. Since the only condition in which seats should be negative is Lunch condition, then it should treat the Lunch condition as though the class was always available. This is the end of modification #1.

Time: 1:55 P.M.

Beginning Modification #2. I am looking at pages 55 and 56 in the documentation because that's where the program is. I am also looking at some sheets of printout produced by the computer at the time that the problem occurred. Sort of keeping my finger in page 55, I am going back in the documentation

to look for the flowchart of this program.

The time is 2:20 P.M. I am going to compare the flowchart on page 31 with the program itself from the auxiliary material that I received. What I plan to do is try to trace the coding with the flowchart and see what the problem is, or if I can spot the problem at all. I suspect that a subscript is allowed to get too large, and I get that suspicion from the error message described on the last page of the auxiliary material. (The error message is, "subscripting destroys define file table.)

According to the flowchart, the first thing that happens is that we read an input card. Starting at the beginning of the program, the first thing that happens is: We do some housekeeping, set some fields equal to zero and the READ statement then is Statement 1180SROF; it's called Statement 1. We read--glancing down through the program. The next thing we have is a header card. And that agrees with the flowchart. It's the first header card in which we set M = M+1, M having been previously set to zero.

Keep going--let me do some writing. Then we set M = 0; M apparently is the counter to see how many student request cards there are. Down here, at Statement 1430, I see subsequent request cards. Statement 15--if N runs 15 is negative 0 or positive--oh, oh, I think that's the problem. Starting from 0, N should be: Going back to the flowchart, I think it <u>should</u> be compared to 14.

Back to page 31. It's not the first request. The number of requests equal to or greater than 15. . . This is testing to see if N is less than 15. Ok. . . if N is less than 15 then we keep going. If it's equal to or greater than, we go to 88, which is a pause statement. Well, actually that would be the 16th card. I think that's the problem. I think the problem is Statement #1450.

OK, I made the modification. I changed statement 1450 from "IF (15), 16, 88, 88" to "IF (14), 16, 88, 88." This is the end of Modification #2.

This is the beginning of Modification #3. I am going to turn in the documentation to the record layout where the MCS are filed and see if it contains the field we are looking for. Figure 3, page 7. Now I have to turn back a page to see how many digits there are in that field. According to Figure 1, it's a four digit field. That's interesting . . . I was looking at the wrong thing. I was looking at the section number which is a four digit field instead of the number of sections. The number of sections is a three digit field.

I am now going into the documentation to see if I can find the flowchart which is the algorithm for this program. OK I have gone to page 29, figure 7. I did not learn anything from page 29 figure 7. I am now going in the documentation to the program itself, which is entirely on page 51. Just flipped back to page 29 figure 7 to look at the flowchart again. Looking at the wrong program. I think I should be

looking at the scheduling program. I'm going now to look at the scheduling program of the documentation on pages 59, 60 and 61. I know what the problem is. The number of sections should not be generated. It simply should be picked up from the master course file. It shouldn't be generated at all.

I am going now to page 59 through 61 of the documentation. I am going to look at the program to try to find how the number of sections is being generated at the present time. The reason for this is, I have to find out how it's being generated now so that I can change it.

I've looked at the program for about a half an hour now and I can't find anyplace where ISEQ (which is the number of sections) is used. It's read in, but I can't see where it's used. And I also can't see where the number of sections is being calculated or is used in any way.

I guess the next thing I have to do is go through the program and identify each variable name. There are a lot of names in here I do not understand.

I think I have found the area where the number of sections is calculated. Starting with statement 56, that's the statement number: IREC = IREC + NIND. Now I'm going to go and try to find out NIND is. I'm looking on page 10, figures 4 and 5, at the FORTRAN names to see if NIND is any field in the disk records. Ok . . it is not there; it is not on page 10.

I am looking now at page 6, figure 1 and 2, to see if NIND is a FORTRAN name on these pages. It is not. This was helpful in that it now appears that NIND is not a data name, it

is not an input field name. I shall have to look for it some place else. (The experimenter instructed the programmer to cease work on Modification #3 at this point.)

This is the beginning of Modification #4. I have just read the modification. At this point my concept of the actions that will have to be taken is: Either the subscript KZ, or the subscript IZ, or the subscript JZ is allowed to get out of bounds.

I am now going to look on page 59 of the documentation. I'm looking at statements 2030 and 2070, to see what the upper allowable limits are for the arrays named in the modification. The reason for this is, to determine what the maximum value of the subscript is allowed to be. At this point my concept of the actions to be taken is: to check the upper limits allowable for these subscripts or arrays, and see in the coding itself if they are exceeded.

I have just looked at the two variables in question; the first one being SVCOD, a two dimensional array having limits of 14 and 15; and the other array, statement 2060, the array JARR, having an upper limit of 15. At this point my concept of the actions to be taken is: Somehow one of the array subscripts is getting incremented too far and has to be limited. Looking at this particular segment was helpful because I had to know what the upper limits of the subscripts were before I tested to see if they were going too high.

I am now going to look at the modification again to see which lines of coding contain the errors.

I have just looked at the modification again. My concept of the action to be taken is not changed. Looking at this segment was helpful in that it allows me to find the areas in question in the program.

I am now going to look at statements 2680 and 2610 in the program. These statements correspond to the areas of error as stated in the modification. I am going to look at the coding on either side of these statements to determine if the subscripts KZ, IZ or JZ are getting too large. My concept at this point, of the actions to be taken, has not changed.

I have just studied sections of the program starting with statement 2660 and ending with statement 2750 or 2760. I have determined that the error is almost certainly not with the subscript KZ. I suspect that the problem is with the subscript IZ. Looking at this section of documentation was helpful in that I started out with the possible error in 3 subscripts, and I have now narrowed it down to one.

I am looking now in the program from statements 2690 and 2700. The reason is the upper limit for the subscript IZ in this case is determined by the variable LM, and in statement 2690 the value of LM is determined by the value of the variable L decremented by 1. I have selected this segment over any other because I think that things like variables can get out of hand.

We have the "distant referent" situation here. Now at this point I do not know the value of L. That is what I have to find out. Find the value of L, and then calculate the value of LM at this point, to determine whether the subscript IZ is allowed to get too large.

I have looked at statements 690 through 2750 or so to try to determine the probable value of LM at the point at which statement 2720 is executed. I have not been able to determine the value of LM, which is the upper subscript at this point. By looking at the coding here, I suspect that instead of getting too large, it's getting too small. I suspect that the subscript is being allowed to become negative and I think that's not allowed.

Looking at this segment of coding was not particularly helpful. It was sort of confusing. I did not find what I was looking for, namely the value of L at the time statement 2790, 2690 rather, will be executed.

I have decided on the modification. Between statement 2690 and 2700 I am going to add a statement. I am going to test at that point to see if the value of LM is negative. The reason is that the subscript IZ has an upper limit which depends on the value of LM at that point, and LM was just calculated as L - 1, so theoretically LM could become negative, and negative subscripts are not allowed. So I'm going to insert a test. I am going to say: If LM is negative, I want to go to 68. If it is non-negative, it can't be zero,

and it can't be less than one, so if it's not positive I want to go to 68; and if it's positive, I want to do the next statement. It does not have a statement number, so I shall have to add one.

Now these statement numbers are not in-sequence statement numbers assigned by the program, so I'm going to have to go through the program to see what statement numbers are not being used. That means that I'm going to have to go through and list every statement number that's been assigned, in sequence, to find out what one is most logical to assign.

OK, I have just determined that 45 is not being used. I'll assign 45 to statement number 2700. So then my source code change comes immediately after programmer-assigned 67. Statement #67 says, LM = L-1. Right after that, I'm going to do a test: IF (LM) 68, 68, 45.

I do not expect to test the accuracy of this modification.

The main reference section which caused me to take this action was programmer-assigned #67 (which is compiler-generated statement #2690) combined with the one immediately following it.

This is the end of Modification #4.

# APPENDIX XII

## Guidelines for Keeping a Magnetic Tape Log of Program Maintenance Mental Processes

1. For all remarks that pertain to a specific piece of code, identify that place by the source card number.

2. When you act upon remembered information, state that you are doing so and what it is that you remember. If you later find out that you didn't remember correctly (wholly or partially) point this out when you make this discovery.

3. When you undertake any action, e.g., decide to look up the meaning of a variable,

   ---state why you are doing so;

   ---state the outcome of your search even if the outcome is that you get sidetracked into pursuing some other line of search.

4. Be natural, i.e., work as you normally would if you weren't recording your thoughts. This may be impossible, but try.

# APPENDIX XIII

## Tape Logs of Experimental Programmers

Notes: 1. The tape logs presented here are not exact transcriptions of the magnetic tapes. What is presented is a digest of these tapes with interspersed comments by the investigator and many long passages reduced to a few summary lines. These may be differentiated as follows:

--exact quotes or close paraphrases are delimited by quote marks;

--the investigator's comments and interpretations are enclosed in parentheses;

--programmer questions are underlined.

2. The numbers appearing here refer to the source card line number.

3E.1 First Programmer's Tape Log

This programmer is a CIRAD part-time employee.

3E.1.1 Before Language Tutorial

"First part is just data definition." (Skips down to 134. Then skips over to 178.) "Some ORIF statements. Doesn't make sense." (Skips down to 319.)

(Reads comment at 319): "Jump table for syntax analysis. Where does JMP get set?" (Answer) "Up above at 317, JMP=XPROG. Doesn't allow for that condition in jump table." (She missed first line because of physical alignment of text.)

"Very difficult to follow program and at same time give line numbers--not a normal practice--makes one lose train of thought."

(Back to jump table at 317. Finds if JMP=XPROG, control goes to PROG. at 412): "STACK (XPRG)--that's setting a rule (PROG1). This says we have found a program and we go to POPYES which means we have a success. We go to POPYES from 416. Where's POPYES?"

"POPYES is at 631. And it looks like we build a tree or part

of a tree. Got there with a GOTO. How do we get out of POPYES?" And where do we go after that?"

($A_t$ 663 finds) "GOTO JMPT, beginning of jump table. But how did JMP get set to anything else in the meantime? That looks loopy!" (Amused chuckle.)

"Let's go back to POPYES and see if JMP gets set to anything else." (At 644 finds) "JMP=YEST(STCPNT). At end of POPYES routine (actually a subpart of the MAIN program) line 662, STCPNT is decremented by 1, so the next time it comes through, the jump keeps getting decremented."

(Returns to line 644.) "What is YEST?" (Familiarity with the language would have told her YEST is an array.) Is YEST a rule? (Looks in rules section beginning at approximately 413 and verifies that YEST is not a rule by checking every occurrence of RUL= that she can find.)

(Continuing search for YEST): "Now I will look in data definitions." (At 3): "Not in items." (At 77): "It is an array, a global array."

(Encounters abbreviation RCC in comment on line 77. Does not see it written out in full on line 76.) "What is RCC?" (Searches data declarations fruitlessly. Goes back to POPYES at 631.) "Maybe RCC is just a stack and not a location." (Surmises RCC might just be an abbreviation for something, but isn't sure.)

(Abandons previous tack. Decides to concentrate on rules) "since mod is to add another rule (correct). I think I should become familiar with how rules are made."

"What does the word STACK do?" (Because of language unfamiliarity does not pick up the . preceding STACK which indicates a call on a procedure. Checks other rules and finds they all start "STACK (something). BNF sheet doesn't have STACK on it." (Over-reliance on literal correspondence between documentation and program.)

"Oh! I bet that's for building a stack." (Comparing rules): "Not all of them have a RECO statement (call), not all of them have a FLEX statement (call), but they all have STACK statements (calls) and a 'RUL=" statement."

(Regarding STACK): "Oh, it looks like it's the name of a routine." (Comparison methodology leads her to question why the same statement is found at 414 and 552 in two different rules.)

(Is still concerned about RCC): "Had this experience with programs before: not knowing if something was a data name or a procedure

name--be good to have something to discriminate classes of names."

(Decides on a linear search maneuver.) "Going to 140 and try to identify at least the unique sections. I need an over all view of program at this point. I'm not going to get hung up on details (she does), and I'm not going to try to follow a chunk of data through the program flow. I'm just going to go through and see if I understand what's going on."

"The first thing we come to is the jump table." (Wrong--this is the lexical scan table which has the unfortunate first label JUMP.) "Looks like pages of just one statement." (Correct--but then because of language unfamiliarity becomes confused by all the ORIF clause headings and the END delimiters. Frankly, even for someone who knows the language, this is a pretty rough statement.) "Not sure what section of program this is." (Speculates):

--"Input routine? Nope, can't be." (No reason given.)

--"Initializing?--because we seem to be going in a loop." (??)

--"Housekeeping?--which is initializing counters at the beginning."

"Okay, I'm going to check some things." (At 239): "Don't understand what's going on." (At 253): "I don't understand what's going on--ok--that's checking to make sure we haven't exceeded 48 characters." (Wrong guess--and objectively, a poor one.)

"Bunch more ORIF statements (clauses)--we're checking STATE. We just set STATE back here." (Returns to lexical scan table.) "We check STATE= something depending on the value of IND--whatever that is. Then we come through and check to see what STATE is and change it again. That seems odd." (Cites examples at 153, 172.)

"Ho, ho!" (Discovers the things TYPE is being set to in 256-272 are the names used in rules in FLEX statements, i.e., calls. Checks forward to rules to verify and enumerate matches.)

"What does this mean?" (Doesn't specify the antecedent. Pauses to register a complaint about the arduousness of using the tape recorder.)

(At 306, reads comment): "End of lexical scan--the first part of the program must be the lexical scan. All of this is called MAIN!"

(At 410, reads comment): "Rules for analyzing AGC assembly language--then begin all the rules."

(She notices that the names appearing as XNAME in the jump table match those found as arguments of .STACK in the rules and checks back and forth on specific examples to verify this hypothesis. Decides that XNAME is a) "stack parameter that indicates the next place to go to is procedure (rule)" (NAME) (Then discovers XNAME is really a rule name.)

"How come we go down there?" (She expected the jump table GOTO's to take her further down in the program and not just to the rule section. This is because she failed to discern the recursive nature of the syntax analysis required to build trees and that the only ways out are when POPYES picks up YEST (0) = YEST (1) = XSUCCS or when POPNO picks up NOST (0) = XFAILR. These YEST, NOST settings are made just prior to the first entry to the jump table.)

(Therefore, seeks further examples at random and finds no help from BNF sheets. "Sort of relates to my lexical scan table."

"Okay, going to check some more." (Starts setting up more detailed correspondences between jump table exits and rule name entries: 321 and 412, 322 and 416, 323 and 418. Notes and is confused by jump table entries into the middle of a rule as at PRG1 from 324. She also finds label STS3. in the rules has no corresponding jump table exit. Finally, reaffirms): "This is all syntax analysis with lexical scan coming first."

(Goes to last rule, BSB at 625, and sees that is ends with GOTO POPYES.)

(Goes to POPYES at 631.) "Does something and then at 646 we have (comment) OUTPUT TREE SECTION." (Wonders if this is end of syntax analysis.)

(Notes that 648-652 test what RUL is or was) "and is only interested in XSTL things or XPRG or XPROG. I guess this is where we have to start. I guess if you're going to output a tree you have to have a program--so I guess that makes sense."

"POPNO is in the output tree section. POPYES isn't." (Fails to see that POPYES ends with a GOTO JMPT at end of its output tree subsection.) "That seems odd. Seems they should both be in the same section. Oh, maybe not."

(Finds labels FAILR at 695 and SUCCS at 699, but doesn't indicate she knows how control gets to one or the other. Under SUCCS, she

notes that we WRITE OUTPUT, at 700, and is amazed that this is still MAIN.) "Weird."

(Checks forward to see how far MAIN goes): "to 1024 and then we start STATUS." (She is looking at the annotation in card columns 73-80.)

"What's so different about STATUS?" (Fails to note that this is a closed subroutine-procedure. Jumps back to label FINAL. at 1016 in MAIN. Finds way down to RETURN at 1023. Regarding RETURN): "I don't quite understand that."

"Then PROC .STATUS. This looks like a whole other program. Don't understand STATUS. STATUS ends at 1071."

"Then we have something called GENSYM. I don't understand what that is either. There are no comments. I think there should be a comment here. There should be a comment every time a section changes name. GENSYM goes down to 1162."

"Then WRTOUT. That must mean write output. Don't really understand that either."

(Discovers that every time she section name changes, PROC appears in the left hand margin): "Looks like a paragraph name." (COBOL conditioning. She then enumerates line numbers where she notes this 1163, 1186, 1209, 1218, 1222, 1227. Is surprised that .FLEX and .RECO are so short based on her recollection of Larkin's explanation. Decides she needs more explanation from Larkin)--"especially this bottom part of the program." (Confesses her lack of fundamental understanding.)

(Goes to .MATCH at 1396. Appreciates comment which she finds there but makes no further remarks. Confesses again she doesn't understand what's going on and that this perusal of the procedures has been) "a wild goose chase." (and that all along she has had her finger planted on something up in MAIN)--"oh yeah, the OUTPUT TREE SECTION." (Backtracks.)

(Complains.) "There are too many sections of output, too many WRITE statements. No wonder she (Larkin) ran out of core. There are WRITE OUTPUT statements at 683, 696, 700 (she is now past the output tree section), 723, 725, 742--May be a constraint of the language, but I sort of doubt it--755, 765, 785, 819--
I just like one output routine and one input routine--takes less core--828, 840, 847--all in MAIN, 3 or 4 of them on a page. In STATUS, still a bunch of them (1032 missed), 1034, 1042, 1044. Have a whole routine here called WRTOUT--How come we have all of these WRITE statements outside of this routine?"

"Weird thing: ENDALL pops up from time to time. This should

end the program altogether." (Because of language unfamiliarity she doesn't realize that ENDALL is an END delimiter which closes all open clauses requiring an END delimiter. Finds ENDALL at lines 1192, 1198, 1205. Looks for another place she remembers having seen ENDALL, but tape ends.)

3E.1.2 After Language Tutorial --(≥ 7 Days Later)

(Decides to concentrate on the requested modification, i.e., LFS (<Statement>))

"I'll go through the program to find where other BNF descriptors are and how they're implemented to give me some idea of how to implement the LFS statement."

"First item in BNF table is something called PROG." (Looks first in GLOBAL section--data declarations and presetting--then remembers) "it's somewhere down MAIN--in jump table around 321. Things in jump table determine if JMP is set to one of the BNF statements. Someplace else, evidently, JMP gets set to those things." (Finds at 317, JMP=XPROG.)

"Where else does JMP get set?" (Goes back to the beginning of MAIN--to line 17.) "JMP is declared logical, 15 bits long but is not initialized, that is, at that point, has no value."

(At 101-115, finds all XNAME items are preset.) "Looks like it's for table lookup for numbering the items in the table."

(At 149): "INITL--probably in back--at 1522. JMP is not initialized to anything there. Don't really understand what that does."

(At 150): "Check to see what .GTNCH does--at 1327." (Reads comment and doesn't understand, but notes it doesn't affect JMP.)

(At 151): ".MATCH at 1396." (Reads comment.) "Doesn't do me any good." (i.e., doesn't change JMP.)

(At 152): "What I probably really should do is write a flowchart on this thing. That would help me understand the overall program, but not necessarily help me implement this modification."

(At 317): "Ah, ha! JMP is set under certain conditions." (?) (Confusion): "There are two jump tables, one at 319 and one at 152." (label there is JUMP.) "I wonder if they're the same. Very strange. Oh, that's not a jump table--checking the condition of STATE. I think I remember I goofed."

(At 321): "If JMP = XPROG we go to PROG. Now, somewhere in the program at PROG it should change the value of JMP."

(Beginning at 412 she checks procedure calls to find where JMP is changed):

"At 412 .STACK to 1321--no."
"At 413 .RECO to 1300--no."
"At 413 .FLEX to 1283--no."
"At 415 .FSUB to 1314--no." (But she has missed line 1318 JMP = XX, XX being the first argument of .FSUB. She sees that in every rule, every call on FSUB is followed immediately by GOTO JMPT which takes control back to the jump table, but she can't see where JMP got changed.)

(At 416): "RUL = XPROG and GOTO POPYES." (Goes there.)

(POPYES at 631: Finds at 644, JMP = YEST (STCPNT)) "What happens in the jump table if JMP = YEST (STCPNT)?" (Goes to data declarations to find at 77 that YEST is an array. Finds at 316, YEST (STCPNT+1) = XSUCCS and interprets): "STCPNT is incremented (wrong!) and therefore, line 644 means that we move to the next line of the jump table." (!)

(Launches headlong into making the modification. She has most of the right places in mind but makes an incorrect modification.)

### 3E. 2 Second Programmer's Tape Log

This programmer is a member of the CIRAD technical staff.

(Starts by reading printed material.
Concentrates first on lexical scan printed material and makes sense of it.
Looks at BNF for AGC Assembly Language with aim of inserting as an allowable statement

LFS (<STATEM>) as an alternative in the <STATEM> line

<STATEM>:: = <OPCODE><LIST> | <OPCOD><EXP> |
LFS (<STATEM> | <OPCD>.)

(More desirable is

<STATEM>:: = LFS (<ORDSTM> ) | <ORDSTM>;
<ORDSTM>:: = <OPCODE><LIST> | <OPCOD><EXP> | <OPCD>)

(Confused by (TEMP) because it looks like a BNF literal in definition lines of <OPCD>, <OPCOD> and <OPCODE> used by Larkin to indicate that these are read into program--space considerations.)

(Because he has forgotten what it is he is supposed to do, he looks at a sample assembly listing to note that usual form of desired new statement type takes the form

LFS (DATA↓26↓).

(Concludes BNF is incomplete (he's correct)--because DATA

doesn't appear as a literal in it (wrong reason). "I don't see anything that refers to op codes. I don't know what the allowable op codes are." Feels he will) "have to interrupt the working of the algorithm at a high level rather than at a very fundamental level."

"It's time to look at the program." (Goes through declares to familiarize himself with the program data names and their meanings--almost on a line by line basis. Thus he picks up and considers many things which are irrelevant to the modification. Looks for sense [meaning] in mnemonics [variable names]. Some have, some haven't.)

(Gets into executable code: Begins by reading comments 140-147. Reference to JUMP causes him to look for some which he finds immediately below.
INITL: Guesses what it does and deduces it's only done once since there is no label before it [or on it].
What is happening here is that he makes small local excursions into the code based on what he reads in the comment referred to above but keeps on returning to read the comment. Correctly surmises that this matches logic of lexical scan literature and that SSTB which he puzzled about when reading variable declarations is the system symbol table [which the comment tells him].
Ends read of Comment.)

"INITL; Not worried about this. Probably takes care of itself. Probably nothing to do with the fix."

"GTNCH: Grabbing next character." (Scans forward and decides to use the Name in columns 73-80 to see where it changes from MAIN. Considers each procedure encountered very briefly..

OPCODE: makes guess about START, FINIS and meaning of code--finally gets to GTNCH)

(1343 in GTNCH: Goes back to ascertain meaning and declaration of INREC. Deduces ICH is character and ICHN is next character,
deduces RECLN = record length ,
deduces EOFC = end of file character.)

(Looks back to declarations at 44 for CCPNT--no information. Guesses (wrong) that .INTEXT (really the read routine) is a conversion routine to "translate" the characters.
Deduces functions of BCPNT and CCPNT from logic)

(INTEXT: encounters this and finds out its true function.)

(STRSSTB: encounters line 1426 which deletes '(' and ')' and notes that someone previously had marked this in red with "DELETE"

and that this is sensible in view of the change.)

(Goes on briefly through STRLST, PUTL, DIAGNT, INITL, SPLOUT reading comments where found.)

(Wishes to determine where in the tree building there are syntax tests. He should ignore tree building and concentrate on syntax alone. He wishes to) "Find out where in tree building the LFS options are assembled and modify that to accept a different format." "Will do this by going back to beginning and then locate the exact place I have to modify. After that it should be simple."

(At 150 he notes that he is looking for way to get to EXIT1. This is noted from EOF determination on lexical scan table. He notes that he has never look at MATCH and doesn't know what it does but it not interested in that now. Looking at lexical scan table:)

"STATE = 1, IND = 15 from 15th column of lexical scan table. Therefore IND indicates what type of character has been found by GTNCH. Therefore that's probably what MATCH does because I remember that GTNCH does not assign a value to IND."

(Looks for EXIT1--finds it at lines 306 and reads comment): "Hooray, this looks like an intricate pointer system. I don't think I'm going to be very hep to this." (NOTE: it is precisely the statements 308-317 that shed the most light on what happens in the syntax analysis.)

(Jumps to comment on 319--looks at XNAMES in jump table and deduces that they are constants associated with different tests.)

(He has noted without saying so that JMP has been set to XPROG at line 317. He looks back to declarations to find out what XPROG is (108-115). Verifies they are constants, but doesn't know what they mean.)

(Decides to go on to see operationally how the XNAMES are used, to gain understanding. He goes from 321 down to 412.) ".STACK (XPRG)" (He reads comment at 410, then sees .RECO ('TITLE') and recalls from documentation that TITLE has to be recognized. Therefore, he deduces meaning of RECO but has skipped so far investigating .STACK.)

(Goes to RECO at 1299 to find the meaning of FAIL.
Deduces: FAIL = 0 means success; FAIL = 1 means failure to find argument of RECO.
While he is there--having noted call on FLEX circa 414. investigates FLEX with same meanings for FAIL regarding TYPE.)

(Goes to next page and looks at STACK at 1320 and sees):

"STCPNT = STCPNT + 1
NOST (STCPNT) = XX and sets up a couple of pointers." (But has no idea what this means.)

(Back to MAIN but pauses at POPNO (664) on way back, having seen reference to POPNO in area of MAIN currently under investigation.)

(Sees RCC stack in comments and tries to look it up!! Can't find in area of 36-STCPNT.)

(Goes back to jump table to investigate opcodes.
Confuses OP for expression operators (=, -, *, /) with OPCODE, OPCOD, OPCD at 562.

--Decides to look at POPNO, POPYES again): "Something's going on with that."

(POPYES 631: Reads comment. Concludes since it ends by GOTO JMPT that nothing can be done unless JMP is changed. Therefore, he looks for such. Finds JMP = YEST (STCPNT) at 644: "What's that (YEST)?" (Finds in declarations--reads comments): "That doesn't tell me much."

(Back to main branch off on syntax analysis:
OPA, OPB, OPC, [OP]CODE ..... CD.

--Confesses at this point he is completely lost--can't recognize the "identification logic" of this program and doesn't really know what it produces, i.e., produces a syntax tree whose meaning he doesn't understand.): "Never encountered that. Can't even find a constant like LFS--disturbing."

(Randomly looks at 500-508 and notes that this requires '(' and ')'. (Larkin error) Therefore, he goes to FSUB 1314): "JMP = XX and YEST(STCPNT + 1) = YY. Beginning to make some sense."

(On his way back to MAIN he discovers at 1248 .OPCODE(START, FINIS). Fiddles around, but deduces basic intent and logic of OPCODE. Neighborhood phenomenon: below in .RND finds meaning of CURPC: therefore, in OPCODE, CURPC = CURPC + OPCODEB(A) means that OPCODE(A) contains the byte length of the instruction having mnenomic in OPCODN(A).)

(Goes to array declarations to find OPCODN but doesn't see where it's loaded. Therefore, he guesses that it is probably set up in INITL which he goes to see): "I hope it's not read in."

(INITL 1531): "Oh my goodness! As I feared--they are read in." (Chases to declarations for meaning of OPCODO and finds out they are opcodes in octal for generating AGC code. Therefore, he concludes, correctly, that LFS isn't RECOgnized but is found by

branching to the OPCODE procedure.)

(Decides he has to find where the OPCODE routine is called from. Starts at 408 and gets to CODEA 585. Assumes that an LFS will have been found. Therefore, FAIL = 0 and RUL = CODEA.
He fails to note that this is a call on .OPCODE with START = 0 and FINIS = 6, meaning search of 0 through 5 inclusive, and therefore he must know if LFS is included in the first 6 OPCODN's.)

(Goes to POPYES 631 and eventually at 663 finds GOTO JMPT.)
"Apparently the setting of JMP is crucial here: it is an indicator of where to go next, that is, JMPT uses JMP to go to another part of the syntax analysis."

(Briefly looks at TREE arrays -learns nothing.)

(Believes he sees recursive nature of this): "300 or so JMP initially = XPROG →PROG and asks the question 'Do I have a program?' if the answer is no for various reasons, so then it goes and asks ' what is a program made out of?' Well, a program is made out of statements. 'So do I have a statement?' So go to XSTAMENT or something like that and then for a statement'do I have some opcode?'
And only when it chews up the whole bunch and puts it in a tree with a yes condition and this big tree represents program--only then will it drop through and then do something. And that's in POPYES in the OUTPUT tree section."

" If RUL = XSTLX, XSTL, XPRG, or XPROG"

(Looks at logic in output tree section and notes .WTREE must mean write tree but doesn't exit there but goes back to JMPT); "How do you get out of this program? Is that a wrong assumption? I think I'm on the right track when I say it's stacking everything up in the tree array recursively until it has everything well defined. It seems it should have everything then ready for assembly, but I don't even see that."

(Looking farther down--encounters WRITE OUTPUT at 722; "Where do we branch to that? Good assembly."

(Confusion: sees RETURN statement after all of output at 1023 and therefore, MAIN is a 'perform' (a la COBOL).

(Backtracks to Start again at beginning of program):
"INITL, GTNCH, MATCH --JUMP etc. until →EXIT1
EXIT1 → JMPT..... 407 or 408. There is specifically a RETURN.
--Why?? How can it return? It's not called. Can't see how to get out of the tree building to get to the output section."

(END of tape.)

# APPENDIX XIV

## Language Statement Types

## Which Define Conceptual Groups

This appendix presents some of the nearly raw data from the Conceptual Groupings experiment. The material presented was written by the observer after each of a number of experimental sessions. Later it was collated in the form presented here: Observations are listed under the types of statements about which the observations were made, and not under the experiment during which the observations were made.

A bare minimum of editing, primarily to identify the experiments by number, has been done to the observations.

The observations, arranged according to statement type, follow:

(1) IF, ELSE, FOR:

In experiment 12, both the explainer and explainee knew the input and output of the system, and the DATA DIVISION was explained only briefly. The PROCEDURE DIVISION was covered only in generalized statements such as "well, here is where we're error checking the cards and then we go over here to ...", etc. Only one error (IF) path was traced.

In experiment 15, the program was explained in a straightforward manner, and no error paths were followed.

In experiment 11, the IF and MOVE statements were used most in the program and comprised approximately 60% of the statements.

The IF conditions were explained, and the subsequent action of the object program was referenced only as far as the branch-to location. The branch-to path was not followed. For example, in the following IF statement, the READ-DATA location-path was not followed.

```
IF BATCH-ERROR IS EQUAL TO 'YES'
GO TO READ-DATA.
```

In experiment 10, the most common entrance to the program was with either an IF or FOR statement. The most common exit was a GO TO statement. The GO TO statement when used as an imperative usually concluded a tagged region and ended the function being performed. The following is an example of a typical opening, tagged statement:

```
BA1.  FOR E=NENT(FLP)-1, -1, 0 $
```

and the following a typical paragraph ending:

```
GO TO B3.
```

In experiment 9, the conditionalities of IF statements did not seem to require an explanation of the code, i.e, the conditionalities LESS THAN, GREATER THAN, EQUAL TO, etc., were not explained. The following is a compound of an IF statement which was used in the explanation:

```
IF NOT RECEIVER-RECORD AND NOT P-O-RECORD
                       AND NOT DATE-RECORD
    MOVE '1156' TO DUMP-CODE GO TO ABEND-JOB.
```

In experiment 10, the prose was exceptionally good. The programming was done using communication-pool items and tables and so this limited the paragraph and data identifiers to eight characters for paragraph identifiers and four and three, respectively for item and table identifier. This

limitation reduced the ability for the program to be "read". The following typifies statement and paragraph identifiers used in this program. The statement is:

```
BA100.  IF NENT (FLP) EQ 200$
            BEGIN
            OUTIN ($1$) = 'LOC (WFLX1) $
            OUTIN ($2$) = 5 $
            RDSRI = ESRN ($E$) $
            etc.
```

And the prose is:

> "If the maximum length of table FLP will be exceeded by insertion of the new entry required for this event, an information message is logged and the event is skipped."

(This particular case may have been misleading; it is possible to make JOVIAL statements more explanatory. The restrictions imposed upon this coding were severe and may have been the reason why the program statements are not self-explanatory.)

In experiment 14, the sentence structure and alignment of the syntactical combination of words within the sentence played a very important part in the explanation. In one example, the structure was:

```
IF RESPONSIBILITY etc.
IS EQUAL etc.
AND etc.
IS NOT etc.
MOVE etc.
```

In experiment 14, identifiers were used as self-explanatory units when referenced in an IF statement:

```
TEST-CHG-LETR.

  IF THIS-IS-A-CHG-LTR AND LAST-PR-NOT CLOSED

      MOVE '3' TO ERROR CODE

      GO TO FAIL-AUDIT.
```

It is obvious what the tell-tale paragraph identifier TEST-CHG-LETR is going to do. The IF statement is self-explanatory because the referenced data identifiers explain their contents.

In experiment 9, the explainer went through the program in a straightforward method, i.e., he did not, with only two exceptions, take conditional IF statement paths. These two exceptions were as follows:

- When a new purchase order record was equal to a previous purchase order record, the 'normal' path was to process that new purchase order. In the following example, the GO TO RECEIVER-UPDATES path was followed

```
IF RECEIVER-RECORD

   IF PART-PLT-PO OF RECEIVER-RECORD EQUAL TO

    PREVIOUS-PART-PLT-PO GO TO PROCESS-

    RECEIVER-REC

   ELSE GO TO RECEIVER-UPDATES

   ELSE, etc.
```

- If a change in the purchase order had occurred, it indicated that a new P.O. or a change to that P.O. had taken place. In either case, they wanted to close out the old P.O. The following statement took care of that situation.

```
IF CHANGE-OCCURRED GO TO SAVE-PO-RECORD ELSE etc.
```

In experiment 12, the following was a typical statement:

```
IF FIL-TYPE = "S", AND

   IN-TYPE  = "N", "U", OR "D", GO TO S-CARD1.
```

In order to make this statement more readable, it could have been stated as follows:

```
IF FILE-TYPE IS EQUAL TO STUDENT AND INPUT-

TYPE IS EQUAL TO NEW, UPDATE, OR DELETE GO

TO STUDENT-CARD.
```

But, the data division would have to be re-described.

In experiment 9, the paragraph structure and alignment were very helpful in "visually" understanding the operation being performed. The following is an example.

```
IF CHANGE-OCCURRED GO TO SAVE-PO-RECORD

ELSE

MOVE SPACES TO LAST-REC-DATE OF NEW-PO-BUFFER

MOVE ZEROS TO RECEIPT-TO-DATE OF NEW-PO-BUFFER,

                    LAST-REC-QTY OF NEW-PO-BUFFER

GO TO SAVE-PO-RECORD.
```

In experiment 14, the documented prose was a condensed form of the program and was not as clear as the program itself. The following is an example of prose vs. program statement:

Prose

"If input resp code in zero, P.O price/m must equal current standard."

Program Statement

```
IF RESPONSIBILITY OF DAILY-INPUT-AREA
   IS EQUAL TO 'Ø'
   AND PRICE-PER-1000 OF DAILY-INPUT-AREA
   IS NOT EQUAL TO MFCPRICEB
   MOVE '1' TO PRICE-ERROR-IND.
```

(2) DO, PERFORM:

Experiment 11 illustrates the conditional use of PERFORM statements to indicate groups to be explained. Once a subroutine had been referenced by a PERFORM statement, that subroutine was not explained again whenever it appeared.

In experiment 14, the PERFORM statements were such that the paragraph identifiers were self-explanatory and the explainer did not have to access the paragraph being described in the statement. The following are examples:

```
PERFORM READ-DAILY-INPUT.

PERFORM WRITE-ADJ-DAILY-ACT.

PERFORM READ-P-O-BAL-FWD.

PERFORM PROCESS-RECEIVER-INPUT.
```

In experiment 11, the program was explained in a straightforward basis, i.e., no conditional statement exits were followed. The one exception to this case was when the PERFORM statement appeared within the code being explained.

In experiment 14, the program was written in the way that it operates, i.e., the flow of the program went from one paragraph to the next paragraph adjacent directly below it. With the exception of the PERFORM statement, it was not interrupted by any other statements such as IF or GO TO.

In experiment 9, the program was explained in modular form, each module consisting of the statements which occurred between tagged regions. With two exceptions, the PERFORM statement was the only statement that caused the explainer to deviate from the "normal" flow of the programs. These two exceptions were when:

A. The PERFORM statement was the object of an IF statement. For example:

```
IF RECEIVER-RECORD

   MOVE '1245' TO DUMP-CODE GO TO ABEND-JOB

   ELSE MOVE 1 TO NEW-PO-IND

        PERFORM READ-ADA

        GO TO MOVE-NEW-PO-S.
```

B. When the object of the PERFORM statement had been described in preceding explanation, i.e., it had already been described once before.

In experiment 13, PROCESS was explained like PERFORM. For example, reading in a card and checking column one of that card was coded and grouped as:

```
PROCESS-CARD.

   READ CARD-IN AT END GO TO FINISH.

   IF COL-ONE IS EQUAL TO 'L' GO TO PROCESS-LOAD.
```

From experiment 11, the following are examples of PERFORM statements initiating groupings:

- In open program regions when it appeared as an imperative statement.

  Example:

```
PERFORM ERROR-CHECK THRU ERROR-END.
```

- In open program regions when it appeared as the object of a simple relation condition.

  Example:

```
IF FIRST CARD IS EQUAL TO 'YES' PERFORM
    NEW-BATCH-IN.
```

- In subroutines, the statement path was followed under any condition, i.e., the path was followed whether or not the PERFORM statement was imperative, a simple or complex relation condition. The following are examples:

  Imperative

```
PERFORM LIST-GOOD-CARDS
 VARYING CARD-X FROM 1 BY 1
 UNTIL CARD-X IS EQUAL TO BUFFER LIMIT.
```

  Relation Condition (Imperative simple)

```
IF RESPP IS EQUAL TO SPACE
 PERFORM IMPORT-CHECK
  VARYING IMP-X FROM 1 BY 1
  UNTIL IMP-X IS GREATER THAN 15.
```

  Relation Condition (Conditional complex)

```
IF BATCH-ERROR IS EQUAL TO 'YES'
 NEXT SENTENCE ELSE
 PERFORM GOOD-CARD-LIST THRU
    G-C-L-EXIT.
```

It should be noted that the PERFORM statement path was not followed when it was a relation condition (conditional) statement in open program regions and it was the first time through the region. The path was followed on the second or sbusequent description of the region. For example, the following path was not followed on the first-time-explained:

Relation Condition (Conditional)

```
EX.1    IF BATCH-TYPE IS EQUAL TO 'P'
            PERFORM P-O-ACCRUAL THRU P-O-X
            ELSE
            PERFORM REC-ACCRUAL THRU REC-X.
EX.2    IF BATCH-ERROR IS EQUAL TO 'YES'
            NEXT SENTENCE ELSE
            PERFORM GOOD-CARD-LIST THRU
                G-C-L-EXIT.
```

The PERFORM statement path was not followed when the logic of the statement was negative in open program regions. For example, the following path was not followed the first time through:

```
IF BATCH-ERROR IS NOT EQUAL TO 'YES'
    PERFORM GOOD-CARD-LIST THRU
        G-C-L-EXIT.
```

(3) GO TO, EXIT:

In experiment 11, the imperative GO TO and EXIT statement paths were always followed wherever they were encountered during the explanation of the code. The following are examples:

```
EX.1    GO TO READ DATA.
EX.2    P-O-X.
            EXIT.
```

Also, the EXIT statement was used only in subroutines which contained conditional statements imbedded within the routine itself. For example:

```
GOOD-CARD-LIST.

  MOVE 'YES' TO BATCH-ERROR.

  IF CARD-X IS EQUAL TO 1

     GO TO G-C-L-EXIT.

  GENERATE

     etc.

G-C-L-EXIT.

  EXIT.
```

In experiment 15, the following statements specify that if "errors occurred while reading card inputs, (you should) quit now.":

```
IF (IQUIT.EQ.0 OR IBYPAS.EQ1) GO TO 201

CALL ERR (201)

GO TO 90
```

The meaning of the GO TO 90 statement is not obvious, and the explainer had to actually study the code to explain its logic (which, ends up with the statement STOP).

In experiment 13, the explainer covered the areas of the program as blocked statements which terminated with either GO TO or EXIT statements. Since he was not covering the actual statements, he did not explain PERFORM staements but instead explained the function being performed in a generalized manner, i.e., he did not go into the subroutine referenced by the PERFORM statement.

In experiment 10, the explainer did not follow the coding statements. Instead, he explained the regions in general terms; he used such phrases as "so this section of code here deals with ..." or "we go to BA1 to go to the next cycle." The explainee had no trouble understanding the program; she had had similar experience in another simulation system.

In experiment 13, the explanation reflected what appeared to be an excessive number of paragraph statements. These statements might have been avoided with more thought. For example, the following coding:

```
DONE-MOV1.

  MOVE BAR-MARK TO DAYS-PER-YR (INDEX-1).

  ADD 1 TO INDEX-1.  IF INDEX-1 IS GREATER

  THAN INDEX-2, GO TO DONE-MOV 2.

  GO TO DONE-MOV1.

DONE-MOV2.

  EXIT.
```

```
DONE-MOV1.

  MOVE BAR-MARK, etc.

  IF INDEX-1 IS NOT GREATER THAN INDEX-2,

  GO TO DONE-MOV1.

DONE-MOV2.

  EXIT.
```

(4) WRITE, READ:

In experiment 13, the following were typical paragraph or data identifiers:

```
READ-MSTP-CARDS.

JUN-1.

COMPARE-UNIT-PROB.

01   CALENDAR-REC.
```

In experiment 14, the words READ, WRITE, and PROCESS were imbedded and used as a tell-tale tag, and it was not deemed necessary by either the explainer or explainee to go to those paragraphs or subroutines to show the function being performed.

Not only were these tags good explanatory units for the PERFORM statement, they were also very valuable to the explainee in understanding the routine being covered at this time.

In experiment 15, the program made much use of the FORTRAN syntactical statements such as WRITE, READ, DO, etc., which were good explanatory units. However, these statements are limited by their object-labels. For example, the following statement was treated as a unit describing a typical WRITE statement:

```
WRIT (6,20)
```

The number 6 in (6,20) means PRINTER, but this number could vary according to the compiler or machine used. The number 20 in (6,20) means statement 20; the programmer has to go there to find out what format to use.

(4) Comments:

As illustrated above, experiment 15 indicated that without qualifying comments, a FORTRAN program could be difficult to understand.

In experiment 9, paragraph, section, and data identifiers were helpful in "reading" the program. Their contents defined what a data item contained or what a paragraph or section of the program performed.

In experiment 15, the program was used for generating scripts; the inputs were cards. The scripts defined simulated impacts of nuclear weapons. The main function of the area covered read cards and performed a table lookup. The tables were two-dimensional arrays.

The programmer/explainee used many comment cards within the program, thus making the understanding of the program less difficult. For example, the following comments and statements helped in defining an array:

```
CWEAPON TYPE ARRAYS.
   DIMENSION JBURS(26), JYIELD(26)
CJBURS IS BURST PARAMETER FROM DS3
     (1=SURFACE, 2=AIR)
CIYIELD IS YIELD PARAMETER FROM DS3
     (1=SURFACE, 2=AIR)
```

and the following is a program statement:

```
CINCREMENT VALID IMPACT COUNT
   NIMPS = NIMPS +1
119 IF (NIMPS.LT.NIMLIM) GO TO 121
    CALL ERR(119)
    NIMPS = NIMP -1
    GO TO 110
```

(6) TRANSFORM:

In experiment 11, the program was explained in a modular fashion. The "modules" were areas between two tagged locations. The only time this thought-unit was interrupted was when a PERFORM statement occurred or the TRANSFORM statement was explained.

Also, the TRANSFORM statements were usually explained at some length. This was explained as being due to "the way the old machine used to handle data" or as a bookkeeping feature to insure "that there are zeros in there". The following is an example:

TRANSFORM PRICE-UNIT FROM SPACES TO ZEROS.

Security Classification

## DOCUMENT CONTROL DATA - R & D

*(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)*

1. ORIGINATING ACTIVITY *(Corporate author)*
Corporation for Information Systems Research and Development (CIRAD), 401 N. Harvard, Claremont, California 91711

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
N/A

3. REPORT TITLE
A STUDY OF FUNDAMENTAL FACTORS UNDERLYING SOFTWARE MAINTENANCE PROBLEMS: FINAL REPORT APPENDICES

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
None

5. AUTHOR(S) *(First name, middle initial, last name)*
None

6. REPORT DATE
December 1971

7a. TOTAL NO. OF PAGES

7b. NO. OF REFS

8a. CONTRACT OR GRANT NO.
F19628-71-C-0125

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
ESD-TR-72-121, Vol. II

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Deputy for Command and Management Systems, Hq Electronic Systems Division (AFSC), L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

"Problems faced by programmers who must maintain programs someone else wrote" were identified. They were reduced to three fundamental inhibiting factors: (1) the limited rate at which people can make "relevance tests," (2) over-confirmation in clues required before hypothesis-testing, and (3) human vulnerability to distraction and procrastination. Studies suggested collectively by these factors were conducted. The studies (1) ascertained that programmers tend to think in terms of conceptual groupings whose objective identification would be helpful, (2) indicated that it was feasible to trace the path the programmer takes as he prepares to make a modification, and (3) identified a few tentative measures of the degree of maintainability of computer programs.

DD FORM 1 NOV 65 1473

Security Classification